Design for Change

SYLVIA STUURMAN

Design for Change

Open Universiteit
www.ou.nl

Design for Change

Proefschrift

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. mr. A. Oskamp
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op vrijdag 12 juni 2015 te Heerlen
om 13.30 uur precies

door

Sylvia Stuurman

geboren op 14 maart 1956 te Heenvliet

Promotor
Prof. dr. M.C.J.D. (Marko) van Eekelen, Open Universiteit, Radboud Universiteit

Co-promotoren
Dr. B.J. (Bastiaan) Heeren, Open Universiteit
Dr. ir. H.J.M. (Harrie) Passier, Open Universiteit

Overige leden beoordelingscommissie
Prof. dr. E. (Erik) Barendsen, Radboud Universiteit, Open Universiteit
Prof. dr. A. (Lex) Bijlsma, Emeritus, Open Universiteit
Prof. dr. S. (Serge) Demeyer, Universiteit van Antwerpen
Prof. dr. ir. S.M.M. (Stef) Joosten, Open Universiteit
Dr. R. (Ruurd) Kuiper, Technische Universiteit Eindhoven
Prof. dr. S.D. (Doaitse) Swierstra, Universiteit Utrecht

Cover design: Aline van Hoof, Visuele Communicatie, Open Universiteit
Cover photo: Ernst Anepool
Printed by Canon Business Service, Heerlen
ISBN: 978-94-92231-11-6

Contents

1

Introduction

Software systems function in a world that changes ceaselessly. It is inevitable that the requirements for any software system change over time: change is inevitable for software. This thesis explores how change in software may be supported, and how we should teach students to design for change.

The world changes, so software has to change. Software that is not adapted to those changes in the world that are pertinent to it, will be used less and less, and will thus become 'dead'.

Software cannot age, one would say: it consists of logic, and thus is not affected by the wear and tear that we humans are subject to. On the other hand, software that has not been adapted can be called 'old'. Parnas indeed called software that has not been updated to meet new requirements *aged software* [Parnas 1994]. Software can, according to him, age in another way as well, after having been updated to meet new requirements: if those changes do not comply with the original design, it becomes harder and harder to apply new changes. More recently, software aging did get another meaning: it is now considered to be an increase in the failure rate or performance degradation of a system as it executes, which can be due to the accumulation of errors in the system state or to the consumption of resources such as physical memory [Cotroneo et al. 2014].

Our focus in this thesis is on the prevention of software aging in the meaning that Parnas gave: the subject of this thesis is how one can design software in such a way that it is easy to apply changes, without the effect that the software ages as a result of those changes. We are interested, one could say, in eternal youth for software.

In this introduction, we will first explore the concepts of design and change in the title of the thesis. In Section 1.1, we analyse the concepts of software evolution and change in software. In Section 1.2, we describe what we mean by software design. In Section 1.3, we explore some difficulties for education in design for change. We present two strategies for design for change in Section 1.4. Each strategy is explored in an individual section: variation points in Section 1.5 and enhancing changebility

in Section 1.6. In these two sections, we relate the chapters of this thesis to these two strategies. We conclude this introduction with a description of the origin of the chapters of this thesis in Section 1.7.

1.1 Software evolution and change in software

The continual change that software undergoes during its lifetime is generally called *evolution* [Lehman and Belady 1985], and the degree to which it is easy or hard to change existing software is often called *evolvability* [Lüer et al. 2001]. The title of this thesis is *Design for Change*, and it is obvious that change and evolution are closely related concepts, but there is a slight difference between the concept of software evolution and the concept of change in software.

1.1.1 Software evolution

Evolution in biology is the process of natural selection, in which those organisms that are best adapted to the environment they live in, will multiply their DNA better than other organisms: those organisms will produce more healthy, productive offspring than other organisms of the same species. Therefore, every change that is rooted in the DNA of an organism and has a positive effect on the number of healthy offspring, will be present in more organisms in the next generation. In each generation, a certain genetic trait will have a slightly different share, and new traits which are created by chance may gain importance over the years. This is what is meant by the concept of 'survival of the fittest' [Mayr 1997].

Evolution in software has a similar meaning. Software that does not change as a response to changing wishes and needs of users will 'die out' because the software becomes less usable to users [Lehman, Perry et al. 1998] so they will abandon the software, which brings the number of replications of it towards zero. However, there is an important difference between the concept of evolution in biology and in software. Variations in DNA are created by accident, without someone directing those changes (we reject the notion of Intelligent Design [Young and Edis 2006], however attractive the expression is considering the title of our thesis). Those variations in the DNA that contribute to individuals who get more healthy offspring than others, will be present in a higher ratio in the next generation. Variations in software, on the contrary, are created by developers with the purpose of getting the software better 'adapted' to its environment.

At first sight, the field of genetic programming might form an exception. In genetic programming, one forces a program to generate variations of itself, and those variations are evaluated against a so-called fitness function. The variation or variations that excell according to the fitness function are then used as the second generation. Eventually, the variation which perfoms optimal with respect to the fitness function might be found. But even in genetic programming, variations are directed, by the developer, to a high degree [Affenzeller et al. 2009].

Software evolution can be observed by everybody. An example is the war (or, using a more positive word, the race) between search engines: Google quickly out-

paced Altavista, which was the main search engine for a couple of years. Since then, many search engines have tried to win terrain, with Microsoft's Bing probably one of the most successful ones [Singhal 2004]. Such a war or race is carried out by applying changes to the software, almost constantly, both with respect to the presentation of the results to the user as with respect to the software forming the results. The user is the 'natural selector' who determines whether a search engine is successful or not.

The changes that are applied are not created at random, but on purpose. Within a development team, there may be team members who analyze how users interact with the product, team members who predict what users might want, team members who formulate new requirements, and team members who translate those requirements into a software design and implement the design. All those activities play a role in software evolution.

1.1.2 Change

In this thesis, we are not interested in deriving requirements or in deciding which changes should be applied; we are only interested in the ease with which changes can be applied to the software. Therefore, we use the term 'change' instead of evolution. We will use the term 'changeability' for the degree of 'easiness' of applying changes to software in such a way that the software will not age, while being aware that this is not a precise definition. For the purpose of this thesis, this definition is sufficient.

The need for change in software is inevitable, because as the world changes, there will be a growing mismatch between the current capabilities of the software and the requirements from the environment. Protocols and standards for communication with other systems might have changed, for instance. This fact, of the inevitable need for change of software, is known as the first law of software evolution [Lehman, Perry et al. 1998].

Changes in software can be categorized along different criteria such as the cause of the change (for instance, a bug or new requirements), the type of change (perfective, corrective, adaptive or preventive), the location of the change, the size of the code modification or the potential impact of the change [Williams and Carver 2010]. In this thesis, those criteria are not important. We are interested in the possibilities to support the future implementation of changes, whatever those changes may be. Because we think it is important to teach our students how to prepare for change in their software design, we are interested in the educational aspect of software change as well.

Design with the intention to optimize changeability is what we mean by the title of our thesis, Design for Change.

1.2 Software design

The design of software should prepare for the accommodation of change, but what is meant by the term software design is not really straightforward. In general, the software construction process is depicted along the lines of Figure 1.1. A software

architecture is drafted, based on the requirements. The architecture is refined into a design, which is the base for programming. The code is built and the program is installed (implementation in Figure 1.1), resulting in the desired system. Software design, in this view, is something between software architecture and programming.

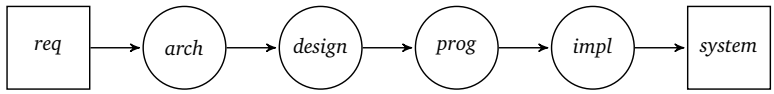


Figure 1.1: From requirements to an executing system

Figure 1.1 might suggest a specific development process, but we do not imply that a specific development process is used, such as the Waterfall model [Royce 1987] or Agile development [Beck, Beedle et al. 2001]. In this thesis, we do not examine the influence of the software development process on the ease to apply changes, even though some of these processes have the explicit goal to 'embrace change' [Beck and Andres 2004].

The word 'design' in the middle might suggest that this is the only activity or product that we are interested in, in this thesis. However, we use the word 'design' in a broader sense. Design in a broader sense, as an activity, means 'to plan and make decisions about something that is being built or created: to create the plans, drawings, etc., that show how something will be made' [Merriam-Webster 2004]. Design in that sense takes place in drafting an architecture, in refining the architecture into a design, and in programming.

The notions of architecture, design and programming usually denote products or activities at different levels of abstraction. However, the boundaries between these activities or products are vague. Also, the words used to discern these levels of abstraction differ: sometimes coding is distinguished from programming; sometimes both are called implementation; sometimes implementation is distinguished from programming (like we do in Figure 1.1). Sometimes design is divided into design and detailed design.

In this thesis, what we mean by 'design', is formed by software architecture, design and programming in Figure 1.1. With implementation, we mean the process in which code is transformed into a running system: compiling, building, installing, configuring.

Maybe the confusion around the meaning of the word 'design' is a result of the evolution of the concept of software design over the years. We present a short overview of this evolution to illustrate the fact that design is involved within each of these levels of abstraction. We only discuss design at different levels of abstraction, and do not discuss (emerging) aspects of design like model-driven development, domain-specific languages, privacy or security by design, and so forth.

1.2.1 Design at the level of programming

Dijkstra was one of the first authors to write about the design of software. The focus was on algorithms: software design for routines (during that time not methods but functions and procedures) [Dijkstra 1969]. Dijkstra advocated explicit reasoning during the design process, deriving the solution from formalized requirements [Dijkstra 1968]. He explicitly stated that this does not mean that there is no design involved: his 'deriving a solution from the specifications' is not to be interpreted as 'automated': software design is a creative process.

Wirth [Wirth 1971], building on Dijkstra's work, introduced the concept of stepwise refinement: define an algorithm in natural language, using functions or procedures that are designed in the next refinement step. His design was at the level of programming, which he contrasted with coding (filling in the blanks in the result of stepwise refinement).

Dijkstra called his method of explicit reasoning and postponing design decisions by making use of subroutines 'structured programming' [Dijkstra 1969]. The goal was to master complexity by designing a program at a slightly higher level of abstraction than lines of code. In structured programming, the software designer solves the problem using abstract machines as building blocks (such as a subroutine). At each refinement step, such an abstract machine is refined until the building blocks are made from constructs of the programming language in use. The abstraction level was shifted from the level of individual statements to the level of subroutines and blocks of statements.

Another aspect of the level of programming is the choice of a specific programming language. By choosing, for instance, a functional language, one chooses a programming style that minimizes side-effects. That means that the choice of the programming language influences properties of the resulting system, as is illustrated, for instance, by a spreadsheet application in Clean [de Hoon et al. 1995].

1.2.2 Design at a higher level

Jackson proposed a graphical notation to support structured programming [Jackson 1975]: Program Structure Diagrams (PSD). He introduced a complementary method to design data structures: Data Structure Diagrams (DSD). He was one of the first to support software design with a graphical notation.

Abstraction thus was the main tool for the software designer to handle complexity. Abstract data types [Liskov and Zilles 1974] are introduced as a set of ready-made abstractions, to be used in different situations.

These ideas on software design fall under the umbrella of modular programming [Gauthier and Ponto 1970], which means that a problem should be divided in separate tasks, that will form separate, distinct program modules. Each module has well-defined inputs and outputs and can be tested independently. Each module can be designed by structured programming. Using modules, one divides a problem into tasks, which is an example of the divide-and-conquer strategy for handling complexity, and modules thus show separation of concerns.

While modular programming has its focus on the functions in the modules, object-orientation shifted the focus onto the data. Objects were introduced: data bundled with the operations to manipulate the data. One of the early examples is Sketchpad [Johnson 1963].

The notion of classes, introduced in Simula [Dahl and Nygaard 1966], was influenced by Hoare's record classes [Hoare 1965]. object-oriented programming languages led to the introduction of (graphical) design languages such as Class Responsibility Collaboration cards [Wilkinson 1998], the Object Modeling Technique [Rumbaugh, Blaha et al. 1990], object-oriented Design [Booch 1982] and the Unified Modeling Language [Rumbaugh, Jacobson et al. 2004]. The driving force of object-orientation was reuse of code.

1.2.3 Again a higher level: Software architecture

Software architecture describes a software system at a higher level of abstraction than functions or objects.

As has been shown, dividing a program into modules has a long history. In 1976, an argument was made for the necessity of a module interconnection language [DeRemer and Kron 1976]. One programming language, was the idea, is needed to program individual modules, and another one to bind those modules together.

Parnas et al. also took the division into modules a step further [Parnas, Clements et al. 1985]. It was argued that software design should divide the software into modules, making use of information hiding and abstraction, and should be accompanied by a 'module guide', a hierarchically structured document describing, at an abstract level, the modules. In the end then, software architecture [Shaw and Garlan 1996] is based on the works of Dijkstra, DeRemer, Kron and Parnas.

When the concept of software architecture gained interest, the emphasis was on the fact that it is possible to look at software systems at a higher level of abstraction the level of abstraction of what is called 'design'. A system could be seen as a configuration or composition of elements, with a rationale. Elements could be modules (which places the module interconnection language [DeRemer and Kron 1976] and the module guide [Parnas, Clements et al. 1985] at the software architectural level), but they could also be data elements, processing elements or connecting elements [Perry and Wolf 1992]. Connecting elements generally are called connectors in literature on software architecture; the term connector instead of connection, points to the fact that connectors are first-class elements that may be composed of other elements [Mehta et al. 2000].

Later on, more emphasis was put on the fact that the high level design of a system can be described from different viewpoints, each view showing certain aspects of the system [Kruchten 1995; Clements et al. 2003; Rozanski and Woods 2005]. One of the possible views is the view of a system at run-time, which is different from the code-oriented, static view of the software design methods that we have described before.

1.2.4 Design in this thesis

What we mean by software design in this thesis, is software design in the general sense. As an activity, software design consists of those decisions that narrow down the solution space. As a product, software design is anything that shows or illustrates those decisions.

Design in this sense is involved in software architecture. Obviously, design is involved in what is generally called 'design' (even though it is not clear what 'design' in the latter sense means exactly). There is also design involved in programming, as is proven, for instance, by the title of the book *How to design programs: an introduction to programming and computing* [Felleisen 2001]. We will refrain from trying to define the fine line between programming (where design is involved) and coding ('filling in the blanks').

This means that we use 'design' in this thesis to denote all products (or all activities) from software architecture to code, where program code has the characteristic that a compiler can derive an executable from this form of design, or that one may run an application using an interpreter that interprets this form of design.

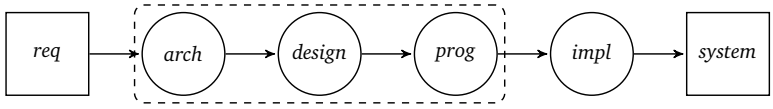


Figure 1.2: Design between requirements and a system

Figure 1.2 shows the meaning of 'design' that we use in this thesis. With the requirements as a starting point, there is always a design (implicit or explicit). Design includes all descriptions of the system-to-be in which decisions have been made about the composition of that system. Requirements are excluded: design will, in the end, deliver something that meets the requirements (or, more precisely, it is the purpose of the designer that the resulting software will meet the requirements). The design is refined until one may implement the system from the code (which may be compiled into an executable or can be interpreted).

What is called 'design' in this thesis, is also called 'software construction' [McConnell 2004]. We prefer the term 'design' in this context: when one buys a box of Lego and follows the instructions, one may construct a car, while the designer of the car is somebody else (the person who created the building plan, and has the job almost every girl or boy dreams about). It is the design we are interested in.

1.2.5 Design principles

The software design principles that have been formulated by the IEEE in the 2014 Software Engineering Body of Knowledge (SWEBOK) [Bourque and Fairly 2014] apply to both the first architectural description of the system and the following refinements. These principles are, so to speak, the mental tools that are available to a software designer. These design principles are:

Abstraction: Abstraction is 'a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information' [Allen, Barnum et al. 2009] and may consist of removing detail (to simplify and focus attention) and of generalizing and identifying the common core or essence [Kramer 2007].

Coupling and cohesion: Coupling is the interdependence among modules (which should be minimized), and cohesion is the degree to which the elements of a module belong together (which should be maximized) [Yourdon and Constantine 1979].

Decomposition and modularization: Software is divided into a number of smaller components with well-defined interfaces. This is usually accompanied by separation of concerns, placing different responsibilities in different components.

Encapsulation/information hiding: Internal details of an abstraction should be hidden, and should not be available to external components. As a result, the details may be changed without affecting any element that uses the abstraction.

Separation of interface and implementation: This can be seen as a form of information hiding: components offer a public interface, and hide their implementation.

Sufficiency, completeness and primitiveness: A software component that is sufficient and complete, captures the important characteristics of an abstraction and nothing more. Primitiveness means that a simple design is better than a complex design.

Separation of concerns: The notion of separation of concerns has been coined by Dijkstra [Dijkstra 1982]. A concern in software architecture is an interest of one or more of the stakeholders of a system, but in the expression 'separation of concerns', a concern is an aspect of the system that is being designed. Separating concerns is a means to handle complexity.

These design principles are not orthogonal. Some principles can be expressed in other principles. Separation of interface and implementation, for instance, may be seen as a form of information hiding or as a form of separation of concerns (one concern being the interface, another the implementation).

Colburn and Shute argue that *decoupling* is at the heart of computer science, as can be seen in the evolution of programming languages: "Stored program architecture decouples programming from physical hardware (wires) [Colburn and Shute 2001]. Formal language translation decouples programming from abstract hardware (zeroes and ones). Structured programming decouples programming from low-level control (gotos and labels)."

Software consists, in essence, of interacting components, and when one component interacts with another, there is a dependency. Such a dependency may form a hindrance when a change has to be implemented, and decoupling means that the

dependency is taken away completely or is made less severe, thus making it easier to implement changes.

In programming constructs or design patterns that aim for changeability, one often sees decoupling. Polymorphism, for instance, means that implementation is decoupled from its interface, and the same applies for the Layers architectural pattern [Shaw 1996] where each layer uses the underlying layer as an abstract machine. The Broker architectural pattern decouples a name from specific components. The Pipe and Filter architectural pattern [Shaw 1996], where each filter does a computation on its input which comes in a stream, and outputs the result in the form of a stream, decouples filters (components) from a configuration (with the result that binding time is deferred). The same kind of decoupling takes place in web services.

Decoupling, thus, plays an important role in design for change.

Another important technique for design for change is *abstraction*, which is, by some, regarded as the key to computing [Kramer 2007]. For instance, the protocol that is the basis of the web, HTTP, abstracts from the type of the content that is sent in response to a request. This abstraction has made it possible to develop plugins for browsers that are able to represent data in new file formats (Flash is an example). Data in new file formats may be sent without any adjustments to the protocol. Abstraction from the content in this example has as a result that the web is open with respect to new file formats and tools interpreting those file formats. Abstraction too may be expressed as a form of decoupling. Abstraction means that something specific (for instance, the content type in HTTP) is decoupled from something more general (for instance, the connection in HTTP).

1.3 Design for change in education

Design for change obviously should be part of the education of a software engineer. More precise: changeability should be one of the important properties of any system to be designed, and in a curriculum for software engineers this should be explicitly taught.

With respect to the education of any subject, there are two questions: what to teach and how to teach it.

The 'what to teach' in this case is a combination of the general body of knowledge of software design, the growing knowledge about how and when to apply decoupling to achieve changeability, and the skill to apply these forms of knowledge in practice.

With respect to the 'how to teach it', one could transfer the idea of an architecture studio, where students work on design projects and learn to become a reflective practitioner [Schön 1987] stimulated by the example of the teacher who reflects on the work of the student, and using student-student interaction as well to stimulate reflection as a habit in the student [Hazzan 2002]. Another approach to 'how to teach it', is to view design for change as a complex task, which should be taught using authentic tasks [Merrill 2002], for which students should receive procedural guidance [Kirschner et al. 2006].

In this thesis, some chapters are dedicated to education of design for change, sometimes focusing on the 'what to teach' and sometimes on the 'how to teach it'.

1.4 Design for change strategies

The area of design for change is a vast area, and also an area in which the chance of finding a 'silver bullet' (a breakthrough that will make it easy to design for change) is close to zero [Brooks 1987]: software design is inherently hard, and software design for change even more so. This thesis, then, does not contain the one key for building software with eternal youth. We offer contributions in several areas of the vast domain of design for change. In this section and the next section, we show how our contributions fit in this vast area.

When considering strategies for design for change, one can discern two strategies to adjust to the fact that the world changes.

The first strategy is to predict possible changes and to provide variation points. Variation points are delayed, and thus open, design decisions [van Gurp et al. 2001]. Variation points therefore, provide places where one can apply a change in the software with relatively little effort. Variation points are a valid strategy when one can predict at least at which places in the software changes will have to be applied in the future.

The other strategy is to enhance changeability: build software in such a way that it is easy to apply changes. In this case, the places where one will have to apply changes are irrelevant: this strategy is also valid in the case that one cannot predict where changes will have to be applied.

These two strategies are not mutually exclusive; one may use them at the same time. These two strategies reflect the dimension of anticipation in a proposal for a taxonomy of software changes [Buckley et al. 2005].

In the next two sections, we briefly discuss aspects of these strategies for design for change and we relate the design principles to these aspects, in particular loose-coupling and abstraction, because, as we have seen, these might be the fundamental principles for design for change. We show which of our contributions fit within each strategy.

Our contributions form the content of this thesis.

1.5 Variation points

An example of a delayed design decision, a variation point, is when the system decides at run-time which exact class will be used to create a certain object. This is possible if other objects that use this particular object only know its type, and not its exact class.

The classes to choose from may be prepared in advance when one can predict all possible situations in which the software could execute. When one cannot predict all these situations, the same variation point offers a possibility to add classes in order to extend the functionality of the system. In the last case, one only has to be able to predict where changes will be needed in the future, and what kind of changes will be needed; in the first case, the changes themselves should be predictable.

When a design decision is delayed until the moment that the system is started, or even until run-time, such a variation point allows for a change in the system after the system has been built.

Part I of this thesis is dedicated to variations points.

1.5.1 Pre-built changes

There are new developments, such as context-oriented programming, to support the construction of software with behavior that depends on its context (such as the hardware it runs on, the location of the device it runs on, available resources, and so on) [Hirschfeld et al. 2008]. The resulting software is called context-aware. Context-awareness is an example of variation points that allow for pre-built variation during execution.

One of the areas where context-awareness of software is very important, is the area of mobile applications (or apps in short) [Dehlinger and Dixon 2011]. There are many examples in which the behavior of an app should depend on the location of the device (an app showing gas stations in the neighborhood is an example). What is displayed, and how, often depends on the resolution of the screen and its orientation. The behavior may depend on the availability of an internet connection (for instance, to store information locally or remotely). The behavior may depend on what the camera sees, and so on. Context-oriented programming has been brought to mobile platforms [van Wissen et al. 2010], but, as is the case with context-oriented programming in general, it has not passed the research phase.

In applications for mobile platforms, one may anticipate many variations that will be needed. These variations, and the possibility that the software reacts to changes of context, should be built in, in advance. For the software to behave differently depending on the context, one needs to apply loose coupling. The operating system for Android mobile devices, for instance, provides a mechanism for communication within and between apps that decouples the communicating partners: intents [Chin et al. 2011].

Because development for mobile platforms, and in particular development of context-aware programs for mobile platforms will rapidly gain importance, it is desirable to include the development of mobile apps in the curriculum of Computer Science. Including the development of mobile apps in the curriculum of Computer Science would possibly have consequences for the software design techniques that we teach our students. As a contribution to this aspect of Design for change, we explore the design of mobile apps, and the possibilties to integrate mobile apps in a curriculum, in Chapter 2.

One often sees a discrepancy between technical possibilities and modeling languages. Some aspects of JavaBeans, for instance, a very simple component model, cannot be modeled using any of the architecture description languages [Stuurman 1999]. In UML, one may model the effects of a certain event on a certain object using a state machine, but it is not possible to relate those events to the classes in a class diagram. Event-based communication is very hard to model using UML [Engels et al. 2000], in contrast to communication by direct method invocation. The same applies to Android intents [Chin et al. 2011] as is explained in Chapter 2.

1.5.2 Design patterns

Design patterns encapsulate variation [Gamma et al. 1994]. Most design patterns offer constructs in which one ensures that an object that uses another object only has limited type information, and does not have to know its specific class. As such, design patterns make it easy to add more classes that adhere to this abstract type. This technique may be used to extend the functionality of a system.

As an example, we applied the Strategy pattern [Gamma et al. 1994] in Figure 1.3, where `TaskList` has an association with interface `Sorter`, to delegate the task of sorting its list-items. In Figure 1.3, there are two classes implementing this interface: `MergeSort` and `HeapSort`.

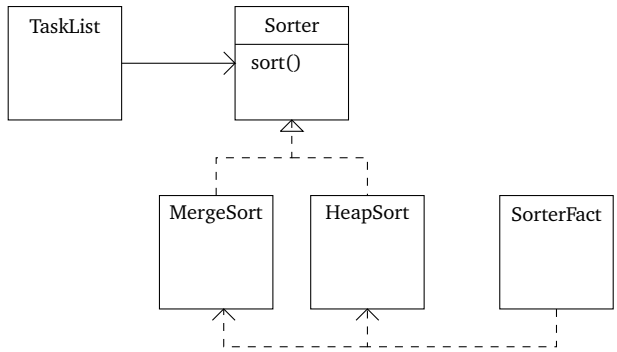


Figure 1.3: The Strategy pattern applied

The essence of this pattern is that `TaskList` never knows the exact class of its sorter; it only knows that this sorter implements the `Sorter` interface. The creation of `Sorter`-objects is done elsewhere, in a factory-class. The Strategy pattern thus is a variation point: the design decision about the specific class of the `Sorter` is delayed until a `Sorter` object is created. When one would like to add another sorter, the only place one has to change is the factory class. The extension point is, in this case, the factory class. By encapsulating variations, design patterns make it easier to apply changes to software.

We see here how loose coupling and abstraction are used to create a variation point. The coupling between `TaskList` and the class of the `Sorter` is loose, which makes it easy to add other kinds of `Sorters`. The loose coupling is implemented using abstraction: `Sorter` is an abstract class or interface.

The problem is that one cannot just loosen every connection and make everything abstract: in the end, there must be places where connections are fixed, because at one point, communication between two objects needs to be established. Also, every variation point adds complexity. The class diagram in Figure 1.3 would be a lot simpler if `TaskList` would just have one method to sort its items. There is always a trade-off.

Creating variation points by using loose coupling and abstraction thus means that one finds places where the current problem shows variability, and tries to predict places where changes will have to be made in the future. Design patterns help to do that: they form general solutions for general problems with respect to variability.

Therefore, a good way to teach students to design for change, is to teach them how and why to use design patterns. They not only guide students in finding variation in a problem and in creating variation points that allow for easy change afterwards, but they also train students in creating software that is flexible with respect to future changes. The problem thereby is that software design is typically something that is learned by doing, by practicing. To experience the advantages that design patterns offer with respect to changeability, one needs to practice with a fairly large system. It is, of course, difficult to offer such practice to students within the constraints of a university curriculum, and this problem is even greater within a distance university.

In Chapter 3, we describe our solution for this problem, using a situation that resembles the software architecture studio proposed by Hazzan [Hazzan 2002].

1.5.3 Dynamic software updating

In some cases, changes have to be applied while the system executes: some systems should not stop, or the occasions in which the system has to be stopped should be minimized. The process of applying changes at execution time is also known as dynamic software updating [Seifzadeh et al. 2013]. In general, such a change is only possible at certain points. Those points are the variation points in this case.

The possibility to delay binding time has increased over the years [Lüer et al. 2001]. At the same time the physical distance (in the same file, in different files on the same file system, on different servers) that is allowed between the elements that, together, form a system, has increased.

Once, one had to put all code in one file and compile that file to produce software. A step further is the possibility to compile code in multiple files into an executable. Again one step further is static linking, that binds multiple files, produced by a compiler, together. Statically linked libraries became available. The files had to reside on the same file system, and a change in a library could only be applied to the system by relinking it: during build time.

Dynamic linking means that a library or module that is compiled may be used by executables when they are linked at start time. A change in a library is thus applied to the system by simply restarting the executable.

One can also use services from one piece of software within another piece of software without linking them: by service reuse, for instance, by remote procedure calls or by web services. Service reuse may be performed across the internet. Binding time, one could say, is during run-time (see Figure 1.4).

Note that increased distance of components not only applies to the distance between the files with code from which an executable is compiled, but also to the distance between the dynamic structures calling each other: this distance increases from within one application to between applications (on the same system or on different systems).

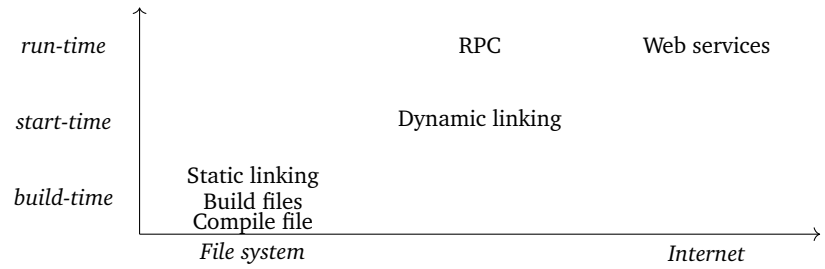


Figure 1.4: Later binding time and increasing distance

Binding time can be seen as variability in time [Capilla and Bosch 2013]. Binding time may occur at design time, compilation time, build time, configuration time, start time or at run-time (in Figure 1.4, we only show a subset). A statically linked library is bound at build time. A dynamically linked library is bound at start time. Delayed binding time is based on loose coupling: when an element is bound after build time, there is no direct coupling from this element with other elements.

Binding time at a later stage increases the ease of applying a change. A change in a statically linked library can only be applied by building the system again, as a whole. A change in a dynamically linked library is apparent in the system as soon as the system is started. The implementation of a web service may be changed at any time: executing systems using such a web service will receive the new service whenever they place a request.

Using the late binding properties of Java, we developed a library with the possibility to 'inject' an executing process with a new version of a class, and to force that process to use that new class instead of the older version. Those classes form the variation points.

That possibility raises new questions. In a system of distributed processes, one has the choice of applying a change either by replacing a single process, or by injecting a new class in such a process. The question is: does this choice have implications on the software architecture in use? What are the consequences of choosing for a replacement of a class versus a replacement of a process?

We explore this question in a case study for a control system for unmanned vehicles, in Chapter 4.

1.5.4 Isolate changes

Another way to create a variation point, is to abstract from the variations and to remove the specific instances of these variations to an isolated place in the code. A factory class is an example: when one needs to add a certain subclass, the only part of the code where a change is needed, is the factory class that decides which specific class will be used for an object.

One step further is to create an Embedded Domain Specific Language (EDSL) with which one may declare the changeable parts of the system [van Deursen et al. 2000].

In Chapter 5, we describe a system for automated feedback on exercises. The feedback engine has been developed in Haskell (a functional programming language) to minimize side-effects, thus enhancing changeability [Jeuring et al. 2007]. In the feedback engine, abstraction is used to make it possible to generate feedback independently from the domain in use (such as proposition logic or linear algebra) and from strategies to solve classes of exercises. Rules for the domain, exercises, and strategies to solve exercises are specified in an EDSL, and the feedback engine is able to use these specifications to generate semantically rich feedback on arbitrary exercises within such a domain. Changes in the domain and in strategies are thus declared outside the main body of the software. The front end, developed by the candidate, uses the same abstraction mechanism to provide a user interface that is geared to the domain of the exercises.

This is an example of using abstraction to make it easier to apply changes: the software abstracts from the specific instances of the variation.

Chapter 5 also describes how we created the possibility to add an infinite number of front ends by offering the feedback engine as a series of web services, thus decoupling the front end from the feedback engine. Both decoupling and abstraction thus are used to create flexibility with respect to changes.

Exercise assistants based on the framework proved helpful to students [Lodder, Passier et al. 2008].

Another example in which specific instances of variation points are isolated, is our JavaScript library for form validation. We developed this library to show how the procedural guidelines that we give our students, lead to elegant code. The starting point is a form validation application for one specific form, and one of the steps that guide students in refactoring their code is to think about possible changes in the future. An obvious change is, in this case, a different form. Instead of having to specify, in JavaScript, which validation function should be used for each input field of a form in the script, one may specifiy everything that is needed to decide which validation function should be used, in HTML. Thus, variations are declared outside the JavaScript code, in HTML.

Chapter 6 describes the set of guidelines, and shows how these guidelines may help to derive elegant code. To declare instances of variation points outside the software is one of the results.

1.6 Enhance changeability

Trying to predict changes, or at least trying to predict where changes will be needed, is one strategy; optimizing the ease of applying a change is another strategy.

The subject of part II is enhancing changeability.

1.6.1 Automatic programming

If one could generate software automatically from the requirements, and one could prove that this translation would produce code that indeed would correctly fulfill those requirements, one would eliminate bugs completely, at any level. The process of automatically generating software is called automatic programming. In 1985, this approach was considered to be at a breakthrough [Balzer 1985].

Parnas [Parnas 1985] on the other hand, argued that, in most cases, it is the solution method, not the problem, for which a specification has to be given, ruling out automatic programming: "In short, automatic programming always has been a euphemism for programming with a higher-level language than was presently available to the programmer".

Automatic programming is definitely possible in certain areas, for instance, for a component that has to check whether the current state of the system conforms to the business rules. Ampersand [Michels et al. 2011], for instance, generates design artifacts such as class diagrams, directly from the business rules.

An example of the ambition of automated programming is the aim to provide a framework for deriving a formal specification of the desired behavior of the system from the requirements (expressed in a formal language), and for automatically translating this specification into Java code, in which every step in this process is proven to be correct [Toetenel et al. 1996].

There are several disadvantages and difficulties with this approach, which we would like to point out.

In the first place, changes should now be applied in the formal specification of the requirements, instead of in the design and/or code. For systems where the requirements should be expressed in a formal language anyway, this is not a problem, but for other systems, it is questionable whether it is easier to change the specification of the requirements according to a changing world than to change the design and/or code. In many cases (for instance in the referenced project) the specification language misses compositionality and other constructs that enhance changeability. This approach thus shifts the problem from designing and coding a software system to expressing the requirements for a system in a formal language. It is very hard to do so for a non-trivial system. For even a modest software system, the number of lines in a formal language would be huge, and it will be very hard to prove that those lines indeed express the requirements [You et al. 2012].

In the second place, changes will always take place during build time with this approach. Applying changes at run-time is very difficult, as the system is created from scratch.

In the third place, it is, in general, not possible to specify non-functional requirements in such a way that code may be generated automatically: non-functional properties are a result of choices in the architecture. Non-functional requirements may be very important.

There is a fourth problem with this approach, that is related to the third problem: in creating the tool that is responsible for generating the code, one makes implicit design decisions, which may have consequences, in the sense of non-functional properties, that one should be aware of. We describe this problem below.

Design in automated code generation One may have the impression that choices are absent in automatic programming, but this is not the case. When one expresses the requirements of a system in a formal language, one makes choices, for instance in deciding which concepts one uses or in deciding how to express those concepts. Other kinds of choices are the decisions that have been made in the framework itself: the choice for the formal language used to express the requirements and the choice for the formal language used to specify the desired behavior. A choice has been made for the programming language, and thus for the programming paradigm. And at last, the software for the translation has been designed: choices have been made there.

The problem is that the results or influences of these decisions on the design are not made explicit (or are even not known). The (architectural) design of a system has an impact on its non-functional properties.

In Chapter 7, we show that the choice for the modeling language for the description of the desired behavior has implications on the design. We describe two software architectures for a railroad controller, with different non-functional properties. We also describe two languages to specify the desired behavior of the railroad system, and show that the choice for one of these languages has implications on the software architecture.

In Chapter 8, we show how we use an explicit architectural style in a framework for distributed control systems with a real-time aspect. The architectural style has known effects on non-functional properties.

These two chapters illustrate the fact that it is better to make implicit design decisions (for instance in a framework for automated code generation) explicit, because these decisions may influence certain properties of the system.

1.6.2 Eliminate side-effects

Maintaining global state in software leads to possible side-effects of any change applied, and unwanted side-effects form a certain type of bug [Kemerer 1995]. Thus, a tactic to prevent bugs when applying changes is to refrain from using a global state. Functional programming is a programming paradigm in which functions have no side-effects, and variables are expressions yielding a value. In most functional languages, one refrains from maintaining a global state.

We already mentioned that the feedback engine we describe in Chapter 5 is programmed using a functional language, thus eliminating side-effects, which enhances the changeability of the software.

Functional programming languages are taught in some university curricula, but they are not very popular outside the university, as can be seen, for instance, in the Tiobe [1] programming community index, which is based on the number of skilled engineers world-wide, courses and third party vendors. There is a development in the direction of functional programming. Programming languages such as C# and Java, for instance, adopt functional features, and the programming language

[1] The TIOBE Programming Community Index, `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

JavaScript has many functional features [Crockford 2008] that are gaining attention from software developers.

There is a problem, however: most professional programmers have experience in procedural or object-oriented programming, and the shift toward a functional style is a difficult one. A key to this shift could be the programming language Scala [Odersky et al. 2010], which is both an object-oriented language and a functional language. Scala is fairly easy to learn for programmers who are familiar with Java or C#, and it advocates the use of functional style programming. Scala thus, could ease the shift between object-oriented and functional programming.

As an aid to an adoption of the use of functional programming, it is therefore desirable for an institution as the Open University of the Netherlands, with a mission around life-long learning, to offer a course about Scala, introducing both the object-oriented and the functional aspects of the language, and teaching how the functional style helps to prevent side-effects.

In Chapter 9, we describe why it is difficult to adopt such a course within the curriculum, and how we succeeded to produce the course in the form of Open Educational Resources. Our method to do so is an example of enhancing changeability outside the scope of software: we enhanced the changeability of our curriculum by producing the course.

1.7 Origin of chapters

Almost all chapters of this thesis are based on a single publication; Chapter 5 is a combination of two publications. This thesis is based on nine publications, all published in academic journals or proceedings, and all peer-reviewed. The candidate is the main author of five of these publications, and one of the two main authors of one publication. Of these nine articles, five have been published in 2007 or later, and three are from 2012 or later.

In the previous sections we showed how each chapter fits in the vast area of design for change. Here, we present the origin of the chapters.

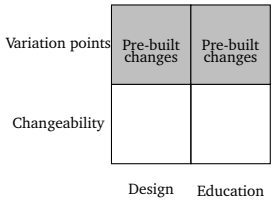
In all publications, we improved spelling and grammar, and sometimes rephrased a sentence to clarify the meaning. In some cases, we did change the order of sections or added text. In the introduction of each chapter, we mention such non-trivial changes.

We provide a diagram with each chapter, indicating whether it is related to variation points or to enhancing changeability(vertically) and whether it is related with design or with education (horizontally).

1.7.1 Variation points

The first part of this thesis contains chapters based on articles that are related to variation points.

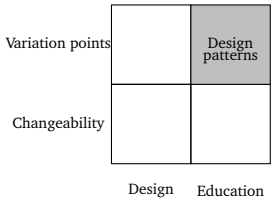
The Design of mobile apps, Chapter 2



This chapter is based on **The Design of Mobile Apps: What and how to teach?**, Sylvia Stuurman, Bernard E. van Gastel, Harrie J.M. Passier (2014) In: *Proceedings of the Computer Science Education Research Conference 2014*, ACM Digital Library. The candidate is the main author.

We explore the constructs of the Android platform that have been developed to optimize the implementation of built-in changes. These new constructs raise the problem of choosing modeling techniques: using UML, for instance, one cannot express Android constructs such as anonymous intents. We explore modeling techniques that might be used, and discuss how mobile apps and these modeling techniques could be integrated in the curriculum.

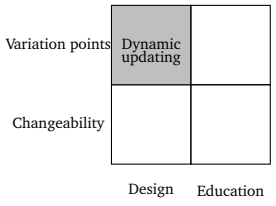
Experiences with teaching design patterns, Chapter 3



This chapter is based on **Experiences with Teaching Design Patterns**, Sylvia Stuurman, Gert Florijn (2004) In: *Proceedings of the 9th annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 151–155, ACM Digital Library. The candidate is the main author.

In this article, we explain that it is difficult for students to appreciate the value of design patterns (and of design for change) by doing small exercises. Only in a fairly big project, one will see the added value of design patterns. We describe how we have succeeded in integrating such a project in the limited time that students have to follow our course on design patterns.

Changes at run-time, at the software architectural level, Chapter 4



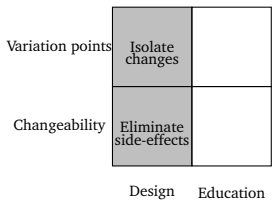
This chapter is based on **On-line Change Mechanisms: The software architectural level**, Sylvia Stuurman, Jan van Katwijk (1998) In: *Proceedings of the 6th International Symposium on the Foundations of Software Engineering*, pages 80–86, ACM Digital Library. The candidate is the main author.

This article discusses the problem of a distributed control system for unmanned vehicles, which is not allowed to be stopped when changes, such as a new version of traffic rules, have to be applied.

We discuss two techniques to apply changes at run-time, which have different consequences for the architectural style in use: one of the mechanisms is in line

with the architecture, while the architecture would have to be adapted for the other mechanism.

Flexible feedback services for exercise assistants, Chapter 5

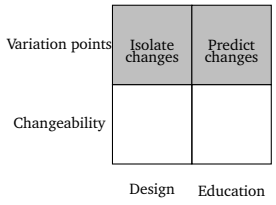


This chapter is based on two articles. The first article is **Feedback Services for Exercise Assistants**, Alex Gerdes, Bastiaan J. Heeren, Johan Jeuring, Sylvia Stuurman (2008) In: *Proceedings of the 7th European Conference on e-Learning*, pages 402–410, Academic Conferences Limited. The contribution of the candidate is in Section 3, Feedback services, and Section 4.1, a client example. The second article is **A Generic Framework for Developing Exercise Assistants**, Johan Jeuring, Harrie J.M. Passier, Sylvia Stuurman (2007) In: *Proceedings of the 8th International Conference on Information Technology Based Higher Education and Training, ITHET*, pages 81–91, IEEE Computer Society Press. The contribution of the candidate is in the Introduction and Section 2.2. The user interface.

In this chapter, we describe a framework for exercise assistants that offer intelligent feedback to steps taken by a student while trying to solve an exercise. The domain, the exercises and strategies to solve them are specified in an Embedded Domain Specific Language (EDSL).

The engine itself is decoupled from user interfaces by offering the functionality in the form of web services.

How to guide students to create good and elegant code, Chapter 6



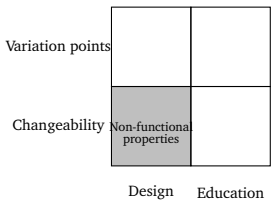
Chapter 6 is based on **Beautiful Javascript: How to guide students to create good and elegant code**, Harrie J.M. Passier, Sylvia Stuurman, Harold Pootjes (2014) In: *Proceedings of the Computer Science Education Research Conference 2014*, ACM Digital Library. The candidate is one of the two main authors. The contribution of the candidate is: the supportive information (Section 3), the procedural information (Section 4), the refactoring (Section 5.4), the evaluation of the results (Section 5.5), the educational background (in the introduction), and the conclusion and discussion (Section 7).

In this article, we describe the guidelines we give our students to refactor JavaScript code into code that is maintainable, and conforms to the SWEBOK design principles. We illustrate the guidelines using a form validation application as an example. In this example, we separate the changes (the specifics of a form) from the JavaScript code: these changes are declared in the HTML file with the form. We also show that predicting changes and preparing for these changes leads to better maintainable code.

1.7.2 Enhance changeability

The second part of this thesis contains chapters based on articles that are related to enhancing changeability.

Software architecture and non-functional properties, Chapter 7

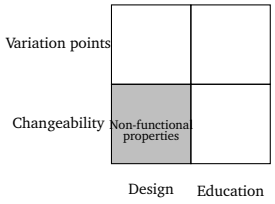


Chapter 7 is based on **Evaluation of Software Architectures for a Control System: A case study**, Sylvia Stuurman, Jan van Katwijk (1997) In: *Coordination Languages and Models, Proceedings of the Second International Conference COORDINATION*, pages 157–171, Springer. The candidate is the main author.

In this article, we explore software architectural styles for a railroad controller.

We make use of two existing specifications of the desired behavior of the controller: an event-action model and an operational model. The event-action model leads to a data-centered software architecture, while the operational model leads to a function-oriented software architecture, with different non-functional properties.

A framework around the radio broadcast paradigm, Chapter 8

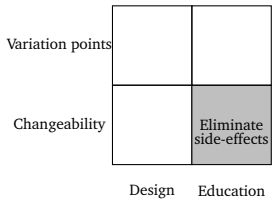


This chapter is based on **Software Development and Verification of Dynamic Real-time Distributed Systems based on the Radio Broadcast Paradigm**, Jan van Katwijk, Ruud de Rooij, Sylvia Stuurman, Hans J. Toetenel (2001) In: *Parallel and Distributed Computing Practices*, pages 105–126, Nova Science Publishers. The contribution of the candidate is in the architectural style and in the overview of implementations of that style: Section 2.1 'Subscription-based communication', Section 2.2 'Existing models and implementations" and Section 3, 'Controlling through subscription: a case study'.

In this article, we describe a framework for the development of distributed control systems with a real-time aspect. We offer an architectural style, the Radio Broadcast Paradigm, with an implementation. We also offer a formal language to describe the system, with the possibility to prove properties of the system, and to derive constraints for the platform on which the system will run.

Creating a short course on Scala, Chapter 9

This chapter is based on **A New Method for Sustainable Development of Open Educational Resources**, Sylvia Stuurman, Marko C.J.D. van Eekelen, Bastiaan J. Heeren (2012) In: *Proceedings of Second Computer Science Education Research Conference*, pages 57–66, ACM Digital Library. The candidate is the main author.



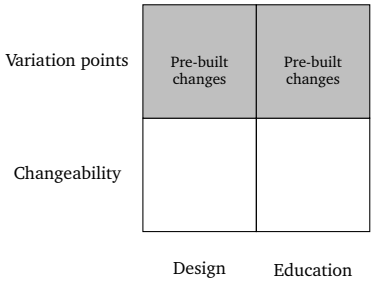
In this article we describe how we produced a short course on the programming language Scala, almost as a side-product of continuous professional development.

Part I

Variation points

2

Modeling techniques for mobile apps[1]



One of the key aspects of mobile applications is that they have to respond to changes in the environment (such as the disappearance of a Wifi or a GPS signal), to changes of the device itself (such as the screen turned from portrait to landscape or the other way around), and to the input of various input sensors (such as a camera or a GPS receiver). Developing platforms such as the Android platform support this type of flexibility. This is a form of pre-built variation.

It is logical, therefore, that a platform for mobile application supports specific constructs and mechanisms, not found in other applications. In this article, we explore the concepts and constructs of mobile applications, and explore modeling techniques that could be used to design a mobile application. The result is a different set of modeling techniques than is, in general, taught in Computer Science and Software Engineering curricula. We discuss various strategies to introduce mobile applications and suitable modeling techniques for the design of mobile applications in a curriculum.

Relevance to this thesis

This article relates to this thesis because mobile platforms are designed with pre-built variation in mind. New constructs in mobile platforms make it easier to build

[1]This chapter is based on 'The Design of Mobile Apps: What and how to teach?' in the Proceedings of the Computer Science Education Research Conference 2014.

applications with pre-built variations, but it is not straightforward which modeling techniques might be used to design such applications. The article also relates to the educational aspect: what should we teach our students with respect to design for change? What are the possibilities to integrate the design of mobile applications in a curriculum?

Deviations from the original article

We changed the order of the sections: Section 2.2 now follows the introduction. In Section 2.4.3, we added references on modeling security aspects. In Section 2.5, we now discuss four models of context-based education, and we did adapt the abstract, the introduction and the conclusions to reflect this change.

Abstract

Mobile applications (or mobile apps or apps for short) gain importance, and will, as is our expectation, find a place in the curricula of Computer Science and Software Engineering. In books, courses and tutorials, not much attention has been given to the design of mobile applications.

In this paper, we describe the anatomy of mobile apps, using Android as an example. Based on this anatomy, we offer an inventarization of modeling techniques that can be applied to adequately design mobile apps. Some of these modeling techniques are already taught in most curricula, albeit in different courses. A modeling technique that is useful for several aspects of mobile apps is the Interaction Flow Modeling Language (IFML). This modeling technique would have to be introduced when one would like to teach students how to design apps.

Mobile apps form a context for various concepts within the subject of Computer Science. There are four models for context-based learning [Gilbert 2006]. We discuss advantages and disadvantages of these approaches.

2.1 Introduction

In courses on Software Design, the domain that is often (implicitly) presumed, is the domain of business applications, running on a server. This is reflected in the examples that are used in textbooks [Evans 2004; Larman 2012].

The omnipresence of mobile applications (or apps for short) shows that the dominance of server-side applications is decreasing. More and more, apps are introduced in Computer Science curricula. Though universities teach knowledge that is as independent of specific technologies as possible, it is inevitable to use specific technologies to teach certain subjects. Both teachers and students prefer recent technology. Because of these reasons, we foresee a shift from the implicit domain of business applications in Computer Science curricula to the domain of mobile applications.

Other reasons are the fact that the user interface of mobile apps has more possibilities than input through the keyboard or the mouse, that mobile applications are often part of a client-server or a peer-to-peer application, that there is hardware involved with multiple sensors, that there is limited memory and there are restrictions

with respect to power usage, that data handling is different than the traditional file-system-model, that interaction between mobile applications is common, and that there are programming issues such as the life cycle of applications, designing for multiple platforms, or security and privacy [Gordon 2013]. In short, mobile applications are rich and complex, and thus suitable to teach many aspects of Computer Science.

There are more reasons. Because of the competitive market it is extremely important in mobile applications to optimize for changeability and adaptability, and therefore mobile applications lend themselves to show the importance of this quality aspect to students. Mobile applications are event-driven, in contrast to server-side applications as they are taught. And, not the least important, learning to program mobile applications motivates students.

The implicitly presumed domain of applications not only influences the examples that will be used, but also the modeling techniques that will be taught. Teaching suitable modeling techniques for the design of mobile applications is important because of the aspects we mentioned before, such as the importance of designing for changeability.

We pose two questions. The first question is: which modeling techniques does one need to design mobile applications? The second question is: if one would like to integrate mobile applications into the curriculum, two strategies may be used to do so. On the one hand, one may use apps as examples in existing courses; on the other hand, apps might become the focus point, to teach concepts and modeling techniques necessary to design apps. What are the advantages and disadvantages of both approaches?

With respect to the first question, we describe the anatomy of mobile apps, using Android as an example. Based on these concepts, we make an inventarization of modeling techniques. Our contribution is a proposal for a set of existing modeling techniques that are suitable to model different aspects of a modern mobile application. We do not restrict ourselves to UML, but also take other approaches into account. Knowing these techniques, a student could model the different aspects of a mobile application.

With respect to the second question, we observe that mobile apps form a context for concepts. We discuss models of context-based learning, taking the fact into account that designing apps may be seen as a complex task, which integrates knowledge from several areas and asks for several skills. We check existing curriculum guidelines for Computer Science and Software Engineering with respect to the models for context-based learning, and discuss advantages and disadvantages.

This paper is organized as follows: in Section 2.2, we describe related work, and show the difference with our work. In Section 2.3, we describe the anatomy of a mobile application. We will use Android apps as an example. Section 2.4 shows which current modeling techniques might be used to model the aspects of apps that we described in Section 2.3. In Section 2.5, we describe advantages and disadvantages of models for context-based learning to integrate modeling apps in the curriculum. We summarize our conclusions in Section 2.6.

2.2 Related work

Research on mobile systems, pertinent to this subject, has been carried out in three areas: research on design methods (often with associated tools) for mobile apps, research on the concepts within mobile apps, and research on how to teach engineering mobile apps.

An example of research on design methods for mobile apps is the work of Parada and de Brisolara on a model-driven approach for the development of Android applications [Parada and de Brisolara 2012]. Here, class diagrams and sequence diagrams of standard UML are proposed to model Android applications. Heitkötter and Majchrzak propose a domain-specific language to model a mobile application [Heitkötter and Majchrzak 2013]. Ko et al. offer an approach in which standard UML is extended using stereotypes, tagged values and constraint meta-classes to model Android applications [Ko et al. 2012]. Kraemer et al. have a different approach. Here, the focus is on the responsive nature of mobile apps; the proposal is to design them using UML Activity diagrams, augmented with State Machines, for which they supply building blocks representing different Android concepts [Kraemer 2011].

What these approaches have in common is that they offer a specific method to design mobile applications, often by using UML. Our focus is different: we try to discern the various aspects of mobile apps and explore which modeling techniques might be useful for those concepts, with the purpose of providing a set of modeling techniques that might prepare students for the design of mobile apps; not by prescribing one method, but by providing students with different possibilities. As far as we know, we are the first to explore the design of mobile apps in this way.

Gordon discusses concepts that are relevant for mobile apps: user interface design and usability, device cooperation, hardware issues, data handling, application interaction and programming issues [Gordon 2013]. His focus is on the knowledge that students need, in different knowledge areas, to be able to create mobile apps, while our focus is on the design techniques they should be taught. Our study complements Gordons research.

Altayeb and Damevski argue that one should teach a model-first approach in developing mobile apps [Altayeb and Damevski 2013]. They use the Prolemy II environment [Davis-II et al. 1999], which offers modeling techniques for Communicating sequential processes (CSP), continuous-time modeling, discrete-event systems, discrete-time, process networks, Petri Nets, synchronous dataflow, synchronous/reactive, and graphics and 3D animations. This is a choice for the first of our strategies to introduce mobile apps in the curriculum: using them as a subject, bundled with the necessary knowledge and techniques. Riley describes how he uses Android programming to teach Java and advanced programming skills [Riley 2012]. Stringfellow and Mule describe the use of an Android project in a course on Software engineering [Stringfellow and Mule 2013]. These researchers have in common that they describe how they use Android or mobile applications in general to teach certain areas of computer science. Our focus is on those modeling techniques that should be taught to students for the domain of mobile applications.

2.3 The anatomy of Android apps

Most mobile platforms support similar constructs though the syntax varies. We use the Android platform to illustrate aspects of mobile applications that deserve attention during the design phase. We choose Android because its open source nature is ideal to study the inner workings.

An Android app runs on the Android operating system in its own Java virtual machine, within its own process, in isolation from other apps.

2.3.1 Elements

Android apps consist of four types of building blocks:

Activities : An activity is associated with a single screen. In texts on Android development, 'design' is often a synonym for the design of the graphical user interface of each screen of an app[2]. Here, we focus on the design of the functionality of each activity and on the functionality of its user interface.

Services : Services run in the background: there is no associated screen. An activity may use a service. Services may perform long-running operations or perform work for remote processes.

Content providers : A content provider manages data. Data may be stored locally (either private for the app or shared with several apps) or on the web. Data may be stored in a file system or in a database. When an application allows other applications to read data, access is through the content provider (unless the data has a simple structure and is private for the app). Activities or services also only read or write data through a content provider.

Broadcast receivers : Broadcast receivers respond to system-wide messages (for instance about a low battery, or an announcement that the screen has been turned-off).

2.3.2 Communication

Activities, services and broadcast receivers are activated through an *intent*. An intent is an asynchronously handled message carrying a characterization of the action to perform (for instance 'start', or 'view', or 'send'), and it may carry a URI or data to act upon. Intents may be anonymous, in which case the operating system searches for an activity or service or broadcast receiver in an app that is able to perform the action, or it may be direct, which means that it is directed to a specific activity, service or broadcast receiver. A result of an intent is delivered in the form of a callback.

Figure 2.1 shows the four types of components. Content providers are the only components that should fetch or save data. Activities, services and broadcast receivers may communicate with content providers through 'ordinary' associations (depicted as solid arrows). Activities, services and broadcast receivers may communicate

[2]Android Developer Guides, Design and Documentation.`https://developer.android.com/`

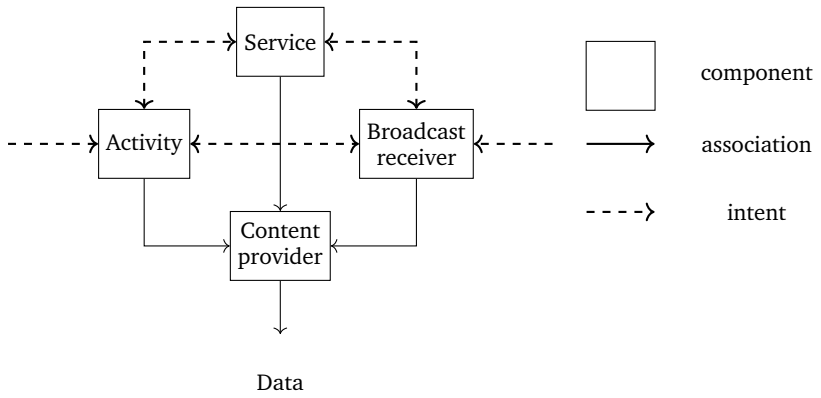


Figure 2.1: The components of an Android app

with each other through intents (depicted as dashed arrows). The system may contact broadcast receivers (for notices) and activities (for instance to start an application, or to ask it to show a video), through intents. Intents may be both directed or anonymous (we have no way to show that in the figure). Intents may be used within one app or between apps. Activities and services may thus send intents to the system or to other activities and services within the app; broadcast receivers and activities may receive intents from the system.

2.3.3 Life cycle

Activities and services go through a *life cycle*, and each of the transitions of one stage in the life cycle to another stage is associated with an *event*. Programming an activity or a service thus means in the first place that one specifies what should be done when each event takes place.

Figure 2.2 shows the states within the lifecycle of an activity. For each state transition, the activity receives an event for which an event handler may be defined, specifying what should be executed when the event takes place.

2.3.4 Threads

Each app runs in a single process and by default all components run in a single thread. However, additional threads may be created, and one may reserve a separate thread for each component of an app. Because the responsiveness of mobile apps is important, separate threads are often needed to avoid a 'freezing' user interface. Services run, by default, in the main thread. Often, one would create new threads

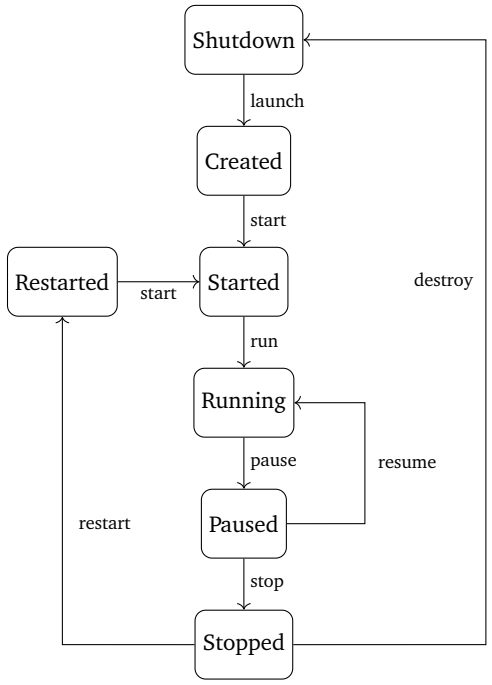


Figure 2.2: The lifecycle of an activity

within a service. The same applies for content providers that store data through the internet.

2.4 Modeling Android apps

Modeling an application before building one, serves several purposes. One may use models to communicate with stakeholders (UML use case diagrams are an example), to have a visual image of the solution at a higher abstraction level than the code (UML class diagrams are an example), to be able to generate code (UML class diagrams are again an example), or, for instance, to be able to analyze the solution with respect to specific properties (Finite State Machines are an example). A diagram technique may be used for one or more of these purposes. Here, we do not discern for which purposes each diagram technique could be used.

Android Apps are object-oriented programs written in the programming language Java and as such can be designed using the standard UML notations as class dia-

grams and sequence diagrams. But mobile applications are often significantly different from standard object-oriented programs. For example, mobile applications are event-driven, mobile applications as embedded software should use limited device resources efficiently, and the development of mobile applications demands additional worries about the short time-to-market. These issues need special attention and the use of specialized design techniques [Kraemer 2011; Parada and de Brisolara 2012; Stringfellow and Mule 2013].

This means that a course about designing and implementing Android Apps requires, among other things, proficiency in the concepts of object-orientation, event-driven programming, and the Android architecture [Riley 2012]. Here, we do not focus on the concepts but rather on modeling techniques. First, we describe the results of interviews we had with app developers about their modeling activities (or absence thereof), then we discuss how to model the anatomy of Androids apps, and finally, we discuss modeling other aspects of Android apps.

2.4.1 In practice

For evaluation purposes we contacted a small number of former students to get information about their experience in developing mobile applications. Although it was by no means intended as an extensive and/or formal survey, a number of interesting patterns emerged. All of them indicated that the user interface (look and feel, to the level of details) is the most important aspect of an app. By using user stories and later on storyboards, they quickly get an idea of what an app should be like. Some use paper prototyping at a regular basis to get the first results. They continue with agile development of the mobile app, and using rapid prototyping methods and A/B testing combined with regular contact with their customers, ensures customer satisfaction. Some indicated that they start with the part of the application that has the highest risk, and continue with reassessing the risk associated with the remaining parts after each step in the development process. In this regard, business decisions are leading in the process.

When asked specifically about their application design, they were at a loss. Their first reaction was to claim that other aspects were more important, i.e. the user interface, the business process, agile development process, and customer satisfaction. As they talked more, keywords as declarative methods, facades, mediators, services, frameworks (for instance, AngularJS or Famo.us) arose, and associated design patterns, the unsuitability of model-view-controller, and the usefulness of the model-view-presenter design pattern. They indicated that although they design an app in their mind, they did not have the proper design techniques to design an application on paper, as they lacked the necessary graphical and semantic representations of important aspects of their application design.

2.4.2 Modeling the anatomy

Activities

Activities are associated with screens. Therefore, use cases can be detailed into a flow of screens, with user input acting as a trigger for a transition to another screen (and thus activity).

One would like to be able to model several aspects of activities that : the functionality, the contents of the screen itself (with a focus on the functionality), the flow of activities and the specification of what to do at each life cycle event. For the functionality, one may use UML class diagrams and UML sequence diagrams, because activities are (subclasses of) Java classes.

Activity Diagrams

The obvious diagram technique to model the flow of activities seems to be the UML activity diagram. UML activity diagrams are meant to model procedural computations, work flows, and system level processes[3]. In an activity diagram, activities may send signals, may wait for signals, and the execution of one activity may lead to the execution of another activity. Activities may be nested.

Activity diagrams may be transformed into Petri Nets for the purpose of analysis [Störrle and Hausmann 2004].

Interaction Flow Modeling Language

An activity is associated with a screen. The flow of activities in an Android app can therefore also be modeled using the (relatively new) Interaction Flow Modeling Language (IFML)[4]. This diagram technique has been developed for the design of (mainly) the client-side of web applications. The most important constructs of IFML are the following:

View container: An element of the interface that comprises elements displaying content and supporting interaction and/or other view containers, for instance a screen.

View Component: An element of the interface that displays content or accepts input, for instance a button.

Event: An occurrence that affects the state of the application, for instance a user pressing a button.

Action: A piece of business logic triggered by an event; either server-side or client-side.

Navigation Flow: An input-output dependency. The source of the link has some output that is associated with the input of the target of the link, for instance the link from a row in a list of artists to a View Component showing information about that artist.

[3] OMG Unified Modeling Language (OMG UML),Superstructure, version 2.2, `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`

[4] `http://www.ifml.org`

Data Flow: Data passing between View Components or Actions as consequence of a previous user interaction, for instance the transfer of information of a shopping cart to a payment action.

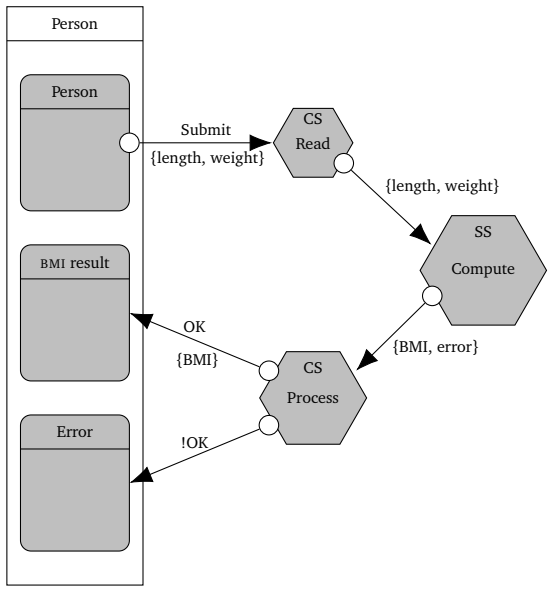


Figure 2.3: An IFML diagram

Figure 2.3 shows an example of IFML used in a design of a web application. We see three view components (`Person`, `BMI result` and `Error`) within the view `Person`. An event (the user submitting a form with his or her weight and length) triggers the activity `Read`, which takes place at the client-side (CS). The event that this activity is ready triggers an activity `Compute` on the server. The event that this activity has been processed triggers an activity `Process` on the client, while the `BMI index` and an `error code` are sent along. At the client, the `Process` activity either produces an `OK` event, which means that the BMI will be displayed in the `BMI result` view component, or an `error` event, which means that the `Error` view component will show an error. An event triggers the replacement. Events may be named, as we have done here for the `Submit`, `OK` en `!OK` events.

As is shown, IFML can be used to describe the flow between screens and the associated activities. A screen may be modeled by a view container or a view component (depending on the level of detail of the model), while the associated activities may be modeled using actions.

There is a UML profile for IFML, which means that it can be adopted by general UML tools; at this moment there is only one dedicated IFML tool available[5].

Services

The functionality of services may be modeled in the same way as the functionality of activities, using UML class diagrams and sequence diagrams. It is not possible, however, to indicate that they run in the background. Often, one would like to be able to model when services perform a certain action (for instance at certain events), and how they use threads to run in the background. Modeling events and threads is discussed below.

Content providers

For the functionality of content providers, the same applies as to services. One could model the type of storage using UML classes.

Broadcast receivers

To model the broadcast receiver, one would like to be able to model which kind of notifications may be received, and which activity or service will handle the notification. As we have seen, IFML could be used for that purpose.

For broadcast receivers, the same applies as to services: one needs to model events and threads.

Communication

As can be seen in Figure 2.1 (where we used 'fantasy' arrows with custom semantics), it is not straightforward how to model communication through intents. One would like to specify whether an intent is directed or anonymous, and what action and which data are associated with an intent. Also, one would like to be able to specify for each activity and service what actions they can perform (for anonymous intents that are associated with actions).

The only research we are aware of, on this subject, uses custom-made building blocks in UML Activity diagrams [Kraemer 2011]. The most obvious choice seems to be an association in a UML class diagram with a classifier specifying that it is an intent, and specifying whether the intent is directed or anonymous. One could also use an association class, to specify the data that are sent with the intent. The arrows in an IFML diagram are very similar to directed intents, as they may carry data. The only restriction is that the arrows in an IFML diagram are bound to events.

At the moment of writing, there is no standard way to model intents. The same applies to message passing in general: it is unclear how to model anonymous message passing, for example.

[5] http://www.webratio.com/portal/content/en/ifml-standard

Life cycle

As can be seen in Figure 2.2, it is easy to model the lifecycle of an activity using a UML state diagram. What is desirable, however, is a means to model the desired actions at each event connected with the life cycle. This might be done by specifying the state the activity has to reach after such an event, but that is difficult. For instance, in a mobile app, there is no guarantee that an event handler is executed when an app stops. One would like to be able to specify what one would like to *try* to achieve at a certain event.

The life cycle of Android apps and the fact that an app may receive broadcast messages are specific cases of the fact that Android apps are event-driven. There are two aspects of events that one would like to model:

- A specification of the state transition that is triggered by an event: the state before the event has taken place, and the state after execution of what is triggered by the event. This is a declarative specification of event handling, focusing on what should be achieved.

- A specification of which actions to take at a certain event. This is an imperative specification of event handling, focusing on how one achieves what is specified above.

With respect to the first issue, the main difficulty is that the state may have been altered between two related events, for instance between the event that an Ajax-call has been made and the event that the response is received. For these issues, Finite State Machines (a graphical notation with a corresponding algebraic notation that lends itself for analyzing [Magee and Kramer 2006]) are a usable modeling technique [Marchetto et al. 2008; Altayeb and Damevski 2013]. Instead of Finite State Machines, one may use UML State-charts, which can be translated into Finite State Machines for analysis purposes [Drusinsky 2011]. Another possibility is the use of Petri Nets [Benveniste et al. 2003]. One has to bear in mind that there is no guarantee, on the Android platform, that the next state is reached after a certain event.

With respect to the second issue, one could use activity diagrams, and – again – IFML. With IFML, it is possible to specify which action is triggered by which event, and one may specify which data are transmitted as well. Handling broadcast messages, for instance, can be modeled in detail using IFML, by specifying the data of a message and the action or service to trigger. The same applies to life cycle events and events triggered by sensors of the device (either detecting user input or a change in the surroundings).

Threads

Because responsiveness is important, one would like to model which functionality is processed in a separate thread. Of course, possible problems with threads and state should be prevented by modeling threads and state, and analyzing those models.

Threads are necessary in event-based systems that must be responsive. The main thread of an Android application is the one that processes UI events, and this thread should not perform heavy computations or long blocking operations in response to user events [Yang et al. 2013]. Threads are notoriously difficult to 'get right', because they are inherently non-deterministic [Liu et al. 2011]. To prevent state-related problems with threads, one may use Finite State Machines, but this can easily lead to a state explosion.

Threads are not only hard to use; it is also difficult to teach concepts around threads, like concurrency and synchronization, in such a way that students really understand these concepts. There have been explicit attempts to focus on these aspects. Li et al., for instance, describe a course on concurrency in which they teach how to use state and sequence diagrams to model concurrent systems. They also show the well-defined transformation from state diagrams to threads-based implementations of monitor constructs and condition variables, and a corresponding transformation to a message-passing implementation [Li and Kraemer 2013].

Because the hard part of using threads lies in the different states that the program may be in, the modeling techniques for threads are essentially the same as for events.

2.4.3 Modeling other aspects

Some aspects of Android apps might be underexposed owing to our focus on the anatomy. We left out, for instance, permissions (which are declared in the manifest file) and security, because modeling security during design is too wide a subject to cover in this article [Myagmar et al. 2005; Lodderstedt et al. 2002; Kou et al. 2010; Mouheb et al. 2009]. Modeling security clearly is a subject that should be taught and discussed in any Computer Science curriculum.

User interaction

The user interface of mobile apps is very important. It must be intuitive and clear, and should be adequate for devices that may differ in the properties of the screen and in sensors for user input. One aspect of user interaction is the visual design, which falls beside the scope of this article; another aspect is the question which kind of inputs each screen should offer, and how these possibilities are associated with the possible flow of activities: this is the question of the functionality of the user interface.

Use Case diagrams

UML use case diagrams are a means for specifying required usages of a system. As such, they may be used to specify which features a user might expect from an app, but use case diagrams cannot be used to model more detailed user interaction.

Sequence diagrams

UML sequence diagrams may model what happens after a user has taken a certain action. As such, they are geared to the specification of a specific interaction of objects, to fulfill a certain use case, but they cannot be used to model more detailed user interaction.

Interaction Flow Modeling Language
IFML cannot be used to model the 'artistic' aspect of the user interface, but it can be used to specify which elements each screen contains (where an element is a view component in IFML terms: an element that displays content or accepts input). Therefore, IFML can be used as the interface between designers and developers: it specifies the functional elements of each screen, which can be used by designers to design each screen.

Distribution of functionality

The functionality of a mobile app may be divided between the client and the server. Some functionality may be implemented on both sides, with a different purpose (for instance, in the case of form validation). The question of how to divide the functionality between client and server is, in essence, an architectural issue. The usual modeling technique to describe these decisions is a UML deployment diagram.

In the case of mobile apps, IFML can also be used to model the division of functionality between client and server. Each action is, by default, placed on the server. An action can be provided with a label `[Client]` or `[CS]` (client-side) to show that it is performed on the client.

Apps may be seen as distributed processes: in general, they have a client process and a server process, and in some cases, there may be a peer-to-peer aspect. This aspect of distributed processes may also be seen in terms of events: a message from the server, from a peer or from another app can be modeled as an event.

User interface design patterns

User interface design patterns play an important role in the design of mobile apps. These design patterns often concern the visual aspects of the graphical user interface. An example is to always display the Cancel button to the left and the OK button to the right [Nudelman 2013]. Other patterns concern the functionality of the user interface (such as asking whether the user has mistyped a search term, instead of assuming a case of mistyping and showing results for what the user probably meant). It would be useful if a modeling technique would allow one to add these patterns as ready-made building blocks for a graphical user interface.

Most patterns for Android are related to the visual aspects of the user interface. There is no support in any diagram technique for these specific design patterns. One can imagine, however, that it is possible to supply building blocks in IFML for some of these patterns. When the screen of an activity should, for instance, contain an OK button and a Cancel button, an IFML building block could refer to the Cancel/OK pattern [Nudelman 2013].

2.5 How to teach

Mobile apps are becoming, and will become, a subject in many curricula. The Curriculum guidelines for undergraduate programs in Computer Science 2013, for in-

stance, have a new (elective) knowledge area 'Platform-based development' for mobile applications [Joint-Taskforce 2013], and the word 'mobile' occurs in various other knowledge areas.

To model Android apps, one has to combine modeling techniques that are associated with different knowledge areas. The Curriculum guidelines for graduate degree programs in Software Engineering [Joint-Taskforce 2009a] mentions Finite State Machines and Petri Nets in the knowledge area of Formal methods, while UML class and sequence diagrams fall under the knowledge area of object-oriented design. The same applies to the curriculum guidelines for undergraduate degree programs [Joint-Taskforce 2013]. When one would have to place IFML in one of the knowledge areas, designing user interaction would be the most appropriate. The design of an Android app thus combines modeling techniques from different knowledge areas. This is in line with our expectation, because mobile apps show concepts from different knowledge areas [Gordon 2013]. Mobile apps form a context for those concepts.

Context-based education is one of the major trends in science curriculum development in the last three decades [Bennett et al. 2007]. The purpose of the use of context is, for instance, to stimulate the transfer of knowledge to other contexts, to be capable of solving problems in various contexts, or to understand the relevance of the concepts with respect to real-life problems [Gilbert 2006]. Another goal of context-based learning is to enhance the attracttion of a subject [Whitelegg and Parry 1999].

There are four models for context-based learning [Gilbert 2006]:

Context as the direct application of concepts: In this model, concepts are central. Concepts are explained as abstractions, and context is used to present examples of those concepts.

Context as reciprocity between concepts and applications: In this model, a context (chosen by the teacher) is used as a vehicle to teach concepts.

Context as provided by personal mental activity: In this model, learners themselves apply concepts to a context, by mental activity. One could claim that the first model is based on the hope that students themselves will use this model to learn.

Context as the social circumstances: In this model, the teacher and the students work together on a task within a certain context. The task involves problems that ask for important concepts.

We will discuss these four models below.

2.5.1 Context as the direct application of concepts

The Curriculum guidelines for graduate degree programs in Software Engineering tells us: 'The principles underlying Software Engineering change relatively slowly, but the technology through which Software Engineering is practiced keeps changing at breakneck speed. Educational institutions must adopt explicit strategies for

responding to changing technology without being caught in the trap of simply training the latest technology. A key to this is organizing the curriculum around enduring principles and planning to change the example technologies regularly" [Joint-Taskforce 2009a]. This makes sense, because the knowledge areas will stay the same during time (while their content will evolve slowly), while application areas or domains will evolve much quicker.

Context as the direct application of concepts translates to the use of mobile apps as examples in courses that are organized around concepts. A clear advantage of this model is that courses within a curriculum may be stable: only the examples will have to be changed from time to time, when new contexts become relevant.

But there are disadvantages as well: "students are not introduced to the social, spatial, temporal framework of a community of practice that is overtly relevant to them; the treatment of the application is usually so sketchy as not to constitute a high-quality learning task; the treatment is not conducted in such a way and for such a time as to allow students to explore the meaning of concepts in the situation presented to them; and what is done does not usually relate to their general background knowledge, beyond that of the concepts that are under immediate consideration" [Gilbert et al. 2011].

2.5.2 Context as reciprocity between concepts and applications

In this model, there should be a cyclical relation between the context and the concepts to be taught: a concept is explained, the application of this concept in the context is presented, a new aspect of the context is discussed to prepare for the introduction of a new concept, and so forth.

There is a big risk that in the end, this model will gradually change into the first model [Gilbert et al. 2011]. Teachers tend to focus on the concepts.

With respect to the curriculum: this approach requires a complete overhaul of the curriculum, when a new context, like mobile apps, is introduced: a specific context covers specific concepts, so the introduction of a new context requires changes in various courses.

2.5.3 Context as provided by personal mental activity

This model does not ask anything of the teacher, or of the curriculum. For students of a distance university, who study, in general, while having a job, often IT-related, this type of mental activity will probably be more common than for students who do not have such background. On the other hand, there is no guarantee that students apply the concepts in the right way, and there is no way for the teacher to guide that process, other than trying to engage them in discussions.

With respect to this model, there are no consequences for the curriculum.

2.5.4 Context as the social circumstances

In this model, the activity of students is to solve problems within the context. They learn concepts on a need-to-know basis.

Problem-solving within the context of mobile applications may be seen as a complex task. An optimal way to learn complex tasks is to base them on real-life authentic tasks [Merrill 2002]. This is because in complex learning, students should integrate knowledge, skills and attitudes. Such an integration does not arise automatically. A focus on learning tasks based on real-life authentic tasks is needed to help learners to integrate knowledge, skills and attitudes [van Merriënboer and Kirschner 2001]. Students should be provided with support and guidance while solving such a task. That guidance should tell students how to recognize an acceptable solution and should provide guidance to the solution process: procedural information is required when one offers authentic tasks to enable complex learning [Kirschner et al. 2006]. This is also in line with guideline 14 'The curriculum should have a significant real-world basis' and 18 'Important efficiencies and synergies can be achieved by designing curricula so that several types of knowledge are learned at the same time' of the Curriculum guidelines for undergraduate degree programs in Software Engineering [Joint-Taskforce 2009b].

A complication with this model may be the fact that it is difficult for students to translate knowledge from one context to another context [Gilbert et al. 2011]. Concepts should be learned within different contexts to overcome this problem, for instance along the lines of the 'concept-context window' [Bruning and Michels 2013].

Support for the task of modeling a mobile app consist of knowledge about the concepts and knowledge about the modeling techniques. Guidance consists of rules to follow: where does one begin, how does one use the different techniques, in which sequence? This kind of guidance should be developed for the context of mobile apps: there is no generally accepted procedure or 'how to' for modeling apps. Also, the adoption of this model would require a complete overhaul of the curriculum.

2.6 Conclusion and discussion

In this article, we addressed two questions: the question of which modeling techniques we should teach our students with respect to the design of mobile applications, and the question of how to integrate modeling mobile applications in the curriculum.

With respect to the first question, we described the anatomy of Android apps, and made an inventarization of modeling techniques for each of the elements of these apps. We did the same for other aspects of Android apps. We showed that the complex nature of mobile applications and the complex concepts that play a role in them, demand skills in and knowledge of a variety of modeling techniques for the design of mobile applications.

Practitioners report that they do not have the proper design techniques, and as a result mobile apps are not explicitly designed in practice. In particular when mobile apps are used as a context in a curriculum, it seems worthwhile to offer students relevant modeling techniques.

When comparing the modeling techniques that are suitable to model different aspects of Android apps with what is taught in Computer Science and Software Engineering curricula, the first observation is the fact that the Interaction Flow Modeling

Language is a 'fit' for several aspects of Android apps, while it is not covered in most curricula (we could not find any curriculum that covered this modeling technique). This means that, when one would like to teach students how to design mobile apps, IFML should belong to the standard set of modeling techniques that are taught to students in Computer Science and Software Engineering. It would be worthwhile to adopt IFML in such curricula.

Other aspects of mobile apps may be modeled using modeling techniques that are, in general, taught within, for instance, courses on object-oriented design and on distributed systems or on formal methods. The IFML could be taught in courses on, for instance, user interface design. While most modeling techniques already have a place in most curricula, the fact that these modeling techniques are spread over such different courses, brings us to the second question.

We have described that modeling Android apps can be seen as a complex task, and should therefore, preferably, be taught as a complex task: using authentic tasks, with support on the concepts and the modeling techniques, and with guidance on how to proceed while designing an app. This is in line with the fourth model of context-based learning: the use of context within a community of practice, to solve problems within that context.

Such an approach is one of three models to teach students modeling techniques for the context of mobile apps (the model in which the student performs the mental activity of applying concepts to a context is irrelevant with respect to the design of a curriculum):

- Teach the different modeling techniques in different courses, along the lines of the above-mentioned knowledge areas: state-related modeling techniques in a course on formal methods, UML in a course on object-oriented design, and IFML in a course on user interaction design.
- Use mobile apps as a vehicle to teach concepts. Focus on designing and building mobile apps from the start, introducing the different modeling techniques and concepts, and gradually make it more complicated.
- Use mobile apps as a context for problem solving, and teach relevant concepts on a need-to-know basis. Provide both knowledge on concepts and guidance on how to proceed while solving the problem.

The first approach is in line with the Curriculum guidelines for graduate degree programs in Software Engineering [Joint-Taskforce 2009a]. All techniques could be integrated in a lab project around an app.

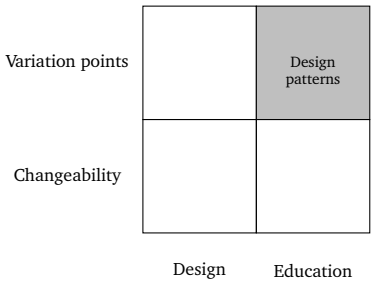
The second approach tends to change into the first appraoch [Gilbert et al. 2011], but has, with respect to the design of a curriculum, the same disadvantages as the third approach.

The third approach is in line with what is needed for complex learning, and is in line with the Curriculum guidelines for undergraduate degree programs in Software Engineering [Joint-Taskforce 2009b]. Adhering to this third approach would mean that the curriculum, in most cases, would have to be overhauled completely. Often, that is impossible.

In either approach, IFML should be introduced, and guidance on how to design mobile apps should be developed.

3

Experiences with teaching design patterns[1]



Design for change is, as we have seen, an inherently creative activity. Therefore, it is hard to teach.

Design patterns are a form of distilled wisdom of practitioners in (object-oriented) design, and can be seen as best practices to design software that is optimized for change. Design patterns help in deciding where and how to apply abstraction and decoupling to create variation points in software. We use design patterns as a means to teach object-oriented design for change, with emphasis on the principles of decoupling and separation of concerns.

In this article, we discuss how we teach object-oriented design using patterns. The problem is that it is hard to appreciate design patterns by doing small exercises: to experience the benefits of design patterns, one needs to work with a fairly large software project. We describe how we have tackled this problem.

Relevance to this thesis

This article relates to this thesis because it discusses the educational aspect of teaching design patterns, while design patterns help in creating variation points.

[1] This article originally appeared in the the SIGCSE Bulletin, volume 36, number 3, pages 151-155, 2004, and in the Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE), 2004.

Deviations from the original article

The main deviation is that related work is now discussed before the main content of the article.

Abstract

In this article, we present an assignment for a course on Design patterns at the masters level, in which students have to adapt an existing program to meet additional requirements. We describe the basic program, discuss the reasons for the decision for such an assignment, and show the results.

The assignment proved to be very effective both to train students to work with design patterns and to assess that students have reached the learning goals. This was true both for students with a professional background and for students with academic interests.

3.1 Introduction

The courses of the Open University in the Netherlands are meant to be studied at home, without extensive help from teachers. In particular during courses at the masters level, we expect students to be able to study independently.

One such a masters-level course focuses on Design patterns. The average student should be able to finish this course within 100 hours. Students are supposed to have a broad knowledge of and experience with Java, to have experience with object-oriented design using UML, and to be able to work with an IDE such as Eclipse and a UML tool such as ArgoUML.

The targeted audience of this course is diverse: on the one hand, we offer a course for students who follow the masters program of Computer Science. On the other hand, Open University students often have a job in the field of their study, so their interest is professional as well in many cases. Therefore, the course should be suitable for software designers who want to gain insight and become better designers: the course should offer professionals usable knowledge and experience.

The course is based on the book Design patterns explained [Shalloway and Trott 2005]. Students read this textbook using an accompanying workbook [Stuurman, Wester et al. 2002] with exercises, background information and explanations. Furthermore, students receive the 'classic' book on Design patterns [Gamma et al. 1994] on cd.

In designing the course, we used several principles and had to meet several challenges.

First, while designing the course and writing the workbook, we held the principle that design patterns are not only learned by reading about them and drawing class diagrams, but by implementing them as well [Beck, Crocker et al. 1996].

Our claim is that this principle of learning by using design patterns helps the academic student to gain insight in the how and why of design patterns, and helps the professional to learn to use them instead of only knowing that they exist. We implemented this principle by providing design and implementation exercises throughout the workbook.

Goldfedder and Rising observed that exposure to a variety of systems was more critical for being able to learn to use design patterns than the number of years of experience [Goldfedder and Rising 1996]. Therefore, we tried to pick examples and exercises from varying domains.

However, this is not enough for both academic students and professionals. Apart from learning individual patterns and the principles behind them, they should learn how to understand and apply patterns they have not seen before, how to integrate different patterns, and how to use this knowledge in real-life situations.

Furthermore, design patterns are not learned by reading and doing small exercises. In small exercises, the task of discerning which problems are present and which patterns could help to solve these problems is a trivial one because of the size of the exercise. Students should train such a competence by working with a fairly large computer program. The same applies to the ability to combine patterns: this is a trivial task in small exercises, and can only be trained truly using a larger program.

Another problem with designing the right exercises stems from the fact that teaching design patterns means teaching advanced object-oriented design. Design can only be learned by thinking about alternatives, and the advantages and disadvantages of these alternatives. This is a difficult task for a student studying at home: a classroom where one can discuss matters with teachers or other students is a much easier environment to train this competence.

Section 3.2 contains some remarks about related work. In Section 3.3, we describe the general idea of our final assignment, which forms a solution for the problems sketched above. In Section 3.4, we describe some details of the program that we use as a base for this assignment. Details about the procedure of the final assignment are given in Section 3.5. In Section 3.6, we summarize the observations we made, while we give conclusions in Section 3.7.

3.2 Related work

Design patterns are often used in introductory courses. They may be used, for instance, to introduce abstractions in CS1 programming courses, to help students reason in a top-down fashion [Wallingford 1996]. Another example is that coding and design patterns are used in programming courses and courses on software design throughout the curriculum, with the idea that design patterns should be teached, and the best way to do so is to introduce various design patterns in various courses. [Astrachan et al. 1998]. A third example is that a set of patterns is used to guide students through the topics of an introductory computer science course [Proulx 2000].

A course at the master's level imposes different requirements, in particular when not only academic students, but also professionals are addressed. The aim is not to teach design patterns per se, but to use design patterns to help students become better object-oriented designers, and to help them grasp object-oriented principles that form the base of design patterns. We have not been able to find other courses than ours in which design patterns are used to teach software design at the master's level.

3.3 The final assignment

The key decision for our course was that the final assignment should consist of a design, an implementation and a report documenting the solution and especially the rationale as to why and how certain patterns were used. In this final assignment, students have to change an existing program to meet new requirements, using design patterns. This means that students can work on a larger program than when they would have to work from scratch. Several sets of new requirements, called scenario's, were defined. The students do their assignments at home, but to force students to think and talk about various solutions, we require them to work in teams of two students.

We do not let the students plunge into the deep at once: before they start with their final assignment, they have worked out two design assignments, both bigger than the exercises in the workbook. In the first design assignment, they are told to try to use a fixed set of design patterns; in the second design assignment, they have to decide for themselves which patterns are useful and why. The second design assignment is done in teams; they tackle the scenario for the final assignment in the same teams.

By offering these two design assignments before the final assignment, we kept to the principle of gradual exposure to complexity [Buck and Stucki 2000]: students start by doing small exercises from the workbook, thus working at the level of knowledge, comprehension and application. After having finished the first half of the textbook and workbook, they do a design exercise using a limited set of patterns, at the level of application and analysis. The next big assignment is again a design problem, this time without a fixed set of patterns, at the level of analysis and synthesis. Finally, the students do their final assignment at the level of synthesis and evaluation, and showing that they are able to find and use patterns not treated during the course.

3.4 The program: Jabberpoint

The starting point for the final assignment is an existing program, inspired by a program written by Ian Darwin: Jabberpoint [Darwin 1999].

Jabberpoint is a slides presentation program written in Java. When we started thinking about a suitable application for the course, we had the following criteria. Ideally, we wanted to use an existing, 'real-world' application instead of a toy program. Furthermore, it would have to be an application that students recognize easily, without having to spend time learning domain details. Likewise, the application should make it easy to think of functional enhancements that could be implemented using patterns. Finally, the program should be small enough to understand in a couple of hours and should be written in Java.

These requirements led us to the idea of using a presentation program for the final assignment. A search on the Internet brought us to Jabberpoint. Jabberpoint was created by Ian Darwin, as a case study for the Java courses he gave. Basically, the program reads an input file (typically an XML file) containing the data for a

presentation, and then displays the presentation in a window, with keyboard and menu controls to nagivate the presentation.

The core concept in Jabberpoint is a `Presentation`. A `Presentation` consists of a sequence of `Slides`, each of which is built up of `SlideItems`. Various types of `SlideItems` are already available, ranging from text to images. Within each `Slide`, the items have a `level` associated with them, indicating the indentation and (font) style properties that should be used in displaying the items.

The display of a `Presentation` is handled in a user interface class that extends `javax.swing.JComponent`. It links to the `Presentation` and keeps track of the current `Slide` in the `Presentation` using the Observer pattern. It retrieves items from the current `Slide`, determines the position for each of them and delegates the actual drawing of the items to the objects of the specific `SlideItem` classes. The user interface is set up fairly implicitly, using `Controllers` (for keyboard and menu) to navigate through the `Presentation` or to load another one.

For the loading and saving of `Presentations`, Jabberpoint contains a number of `Accessors`. Formats supported include XML, HTML and plain text.

Jabberpoint meets most of our criteria. It is both small and big enough, it is written in Java and it provides basic functionality that most of the students will understand. Furthermore, even occasional usage of a presentation program suggests several functional enhancements.

In order to make Jabberpoint usable as the basis for our final assignment, several changes were needed. While considering exercises and (pattern-based) solutions for them, we refactored and simplified the code in some areas. For instance, some classes were renamed (such as `Presentation` instead of `Model`, `Slide` instead of `M`), interfaces were introduced, the `Accessors` were simplified, and the Observer pattern that was already contained in the original code, was written out of it. The task of bringing the Observer pattern back into the program was turned into an introductory exercise for students as a way to get to know the ins and outs of the program.

3.5 The change scenarios

Each team receives a change scenario. Examples of these scenarios are:

- The program should be able to show two extra views on the slides, such as a slidesorter, slides with notes, an outline, or a next slide previewer (a small window showing the next slide).
- It should be possible to have hyperlinks on slides, combined with one or more actions, such as playing a sound, go to the next or previous slide, go to slide number x, open a new presentation, etc.
- While showing a slide, it must be possible to draw on a slide. These drawings should persist while the presentation is shown, but should not be saved permanently.
- Instead of showing all elements of a slide at once, it should be possible to show only the elements of a certain level, or to show them one at a time.

- It should be possible to define more than one presentation using one set of slides. The order of the slides may change, slides may be left out, or may be used more than once.

It is easy to think of more scenarios, so it is possible to have students work at a fresh set of scenarios in the future.

The procedure for the final assignment is that teams first develop a design and send it to their teacher, who comments on the design. Then the teams make their final design, implement it and make up a report, explaining what they changed, which design patterns they used, which alternatives they considered and why they decided to choose for the solution they present.

Teams are asked to describe which work was done by each of the students, and they have to agree on that description. The first 18 students held a presentation about their solution, but we dropped that requirement because of the amount of time it required of the students.

3.6 Observations

At the moment of writing, 38 students have finished the course. The first 18 students who took the course were asked to fill in evaluation forms. We also had a meeting with these students, during which the teams held a presentation about their solution, and told us what they liked and disliked about the course.

3.6.1 Time spent

The first observation concerns the amount of time students needed. The idea was to use 28 of the 100 hours for the final assignment. One of the 18 students did need fewer hours (12); the others did need far more than those 28 hours, with a maximum of 75 hours. The average time needed was 38 hours.

When asked about it, the common answer was that they deliberately spent more time than needed because they became hooked. They tried (and implemented) several solutions to compare, just for the fun of it. The general opinion was that the final assignment could be completed in about 28 hours, under the conditions that the program would be introduced earlier in the course so students would already be familiar with it, and that the presentation of the solution of each team would be dropped.

3.6.2 Working in a team

A second observation is that students preferred working in a team for this course, as opposed to working individually. This is a remarkable fact, because Open University students in general dislike working in teams (or at least say they do), because of practical problems.

Two students did work individually for different reasons; all other students did work in a team. Students discussed several solutions with each other, and commented on having learned from having to give arguments, and being confronted with the ideas of someone else.

One of the students lived in California, one in Germany, eight in Belgium, and the others in the Netherlands. Obstacles for cooperation were time-related (one member of the team studying during weekends, the other one at workdays; one member of the team studying at night; the other one during the day), and not related to the impossibility of personal contact.

The workload was equally divided in all teams, according to the students. In some cases, both students worked out a design, discussed it and chose a final one, then divided the classes to implement and the parts of the report to write; in some cases the work was divided by having one student make the design, having the other comment it, and working the other way around for the implementation.

We tried to avoid the situation that one student does all the work and the other receives a 'free' grade, by asking them to state how the work had been divided, and having to agree about that statement.

3.6.3 Learning design patterns

Because students did have to submit their design, their implementation and a report describing the design and the rationale, we were able to determine that all students had understood the meaning of using design patterns, that they could spot problems where a pattern might come in handy, that they could search for patterns that could provide a solution, and that they could argue why their solution was the most flexible solution with respect to future changes. All students were able to explain which future changes would be easy because of the solution they had decided upon. In other words: all students showed they had mastered the course.

In their comments, many students told us that the two design assignments and the final assignment (in particular the final assignment) had helped them to grasp the concept of a certain design pattern: being confronted with a real problem was what they needed to 'see the light'. Many of the professionals told us they had begun to make use of their knowledge of design patterns in their work during the course: it was very easy for them to see the applicability of design patterns in the context of their work.

3.6.4 Academic students and professionals

Of the 38 students, 15 studied the course for their master's degree; the other students only had a professional interest.

There was no difference between those two groups with respect to the degree of their appreciation of the course. Professionals commented on the usefulness of what they learned for their work; academic students commented on the thoroughness of the material and the depth of their understanding.

3.6.5 Support

We did not offer special support for working within a team, so students relied on e-mail and telephone to communicate. Providing a versioning server (e.g. CVS) for cooperation was considered, but declined because of the troubles getting clients to work on different platforms, and because of the effort for at least some of the students to learn to work with it. The only support we gave students was a proposal on how to divide the work, and tips on how to find a suitable teammate.

Our support proved to be sufficient: working together has worked out very well for all teams.

Another form of support consists of a website for the course, where we have collected links to more information about the topics of the textbook and workbook. The website was used by students during the course, but they almost exclusively used the cd containing the book 'Design patterns' [Gamma et al. 1994] to look for useable patterns for the final assignment.

Students did not use the discussion group that we provided for the course to discuss problems with their assignments with other students. They did use the discussion group during the rest of the course, so the explanation is probably that they believe in an implicit rule that assignments should not be discussed. We will consider the consequences of telling students explicitly that assignments may be discussed in this discussion group.

Teams were also supported by allowing them to send in a draft design to the examinator, who commented on it. We chose for this procedure to help students avoid spending much time in dead alleys. This form of support was highly valued, especially the fact that the comments were given within a day or two after sending in the draft.

3.7 Conclusions and discussion

When one teaches software design by using design patterns, the academic and the professional world almost meet by definition: patterns are a form of distilled professional knowledge, and by studying them at an academic level, the mechanisms of object-oriented design principles become clear. Design patterns have been 'discovered' in the academic world, and are based on thorough professional experience. Teaching software design through design patterns at the master's level, in such a way that the interest of both professionals and academic students are met, requires hands-on experience on a project of sufficient size and complexity.

Our approach of giving change scenario's for a given program has proven to be a solution for the problem of having students work on a fairly big program to learn from practical experience, but at the same time making it possible to finish the course within a limited amount of time. Students were able to finish the course within 100 hours (after some adaptations to get the final assignment fitting within 28 hours), and their reports showed that they did learn what the course tries to establish, for instance, by applying patterns not taught within the course.

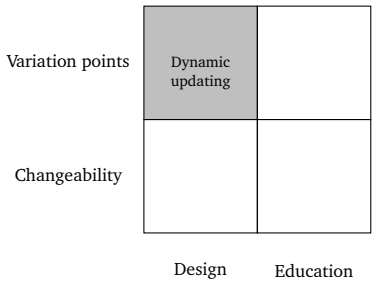
Cooperation within teams had the form of contact by e-mail and telephone, without personal contact, in all cases. The limited support for cooperation proved to be sufficient for all students.

Some considerations when using this approach are:

- Working out a scenario often both consists of refactoring the original program (restructuring without changing the functionality), and adding functionality. In principle, it would be better to separate these two activities. We are afraid that the assignment would cost students more than 28 hours if we would ask them to explicitly separate these activities, but it would be worthwhile to experiment with refactoring before designing extra features.
- Students would experience the benefits of design patterns even more than in the current assignment if they would have to integrate two solutions for two different scenarios. That would show how patterns really enhance flexibility. We cannot take this approach because of the time constraint, but we are considering these type of assignments in courses building on the knowledge learned in this course.
- The two students working alone sent in poorer results than all students working in a team. Of course, the numbers are not high enough to make statistically sound conclusions, but the prediction that working in a team, and therefore being forced to discuss solutions with each other benefits the learning process at least seems to hold. It is strongly recommended to have students work in teams.

4

Changes at run-time: The software architectural level[1]



As a historical note, in the original article we use 'on-line change' to refer to change at run-time. Because 'on-line' is now associated with 'connected to the internet', we have changed 'on-line' into 'at run-time'.

We present a software architecture for a control system of unmanned vehicles, based on processes communicating through subscription-based communication. Subscription-based communication allows processes to stay anonymous. We present two different ways to change such a system at run-time: by adding and/or removing processes, and by injecting changed classes into the system.

The first mechanism is to replace components in the architecture: processes. This mechanism demands a direct connection between the process inducing the change and the process to be replaced, thus breaking the architectural style.

The technique of the second mechanism, replacing Java classes at run-time, was developed by us. It is possible to implement this mechanism while adhering to the subscription-based style.

We discuss advantages and disadvantages of both mechanisms. These two mechanisms are an example of change in an executing system at the architectural level,

[1]This article originally appeared under the title of 'On-line Change Mechanisms: The software architectural level' in the Proceedings of the 6th International Symposium on the Foundations of Software Engineering, pages 80–86, 1998.

which makes it easier to apply changes without breaking the architectural style [Stuurman 1997].

The – maybe counterintuitive – outcome of the experiments is that in this case, changing the system by replacing components is only possible by breaking the architectural style, while applying a change by forcing processes to use a new Java class instead of the original one is possible while adhering to the architectural style.

Relevance to this thesis

The link with this thesis is the fact that it is about dynamic software updating. The system is a distributed system of unmanned vehicles. Changes that are necessary from time to time, without stopping the system, are new traffic rules for these vehicles. We show the consequences for the architecture of the choice of variation points: processes or Java classes.

Relevance today

Dynamic code evolution, the technique to update programs at run-time, is still not possible in statically typed languages like Java without providing supplemental techniques. Techniques for exchanging a class with a new version at run-time are still being developed regularly [Pina and Hicks 2013; Würthinger et al. 2010; Gregersen and Jørgensen 2009]. We did not find other research than ours on the implications of this type of change at run-time on the software architecture.

Software architectures that allow for dynamic updating in general often use a choreographer, a central component with connections to every component that may be updated dynamically [Cîmpan et al. 2005]. More in line with our dynamic software updating using new versions of classes is a framework that makes use of existing possibilities of 'hot deployment' such as found in J2EE, and assures that the result still adheres to the software architectural style [Song et al. 2011]. In contrast, our technique of injecting new versions of a class assures adherence to the software architectural style without the need to check such adherence at run-time.

We do not know other studies in which different techniques for dynamic software updating are compared with respect to the software architectural styles these techniques adhere to. Most studies focus on the problem of changes that have an impact on the software architecture: evolving software architectures [Stuurman 1997; Barais et al. 2008].

Abstract

Our interest in the field of software architecture is focused on the application in technical systems, such as control systems. Our current research in this field is centered around a real-life case study: a control system for unmanned vehicles transporting containers on the 'Maasvlakte', an area in the ports of Rotterdam.

Important issues in this control system are scalability, evolvability, and run-time change capabilities.

In this article, we discuss two mechanisms for change at run-time in a distributed control system for the Maasvlakte system, which we have implemented in

Java. The software architecture we use is a configuration of distributed processes, communicating according to the subscription model.

We will focus on the software architectural aspects of the mechanisms for change at run-time. One of these mechanisms is associated with the decoupling of processes as a result of the subscription-based communication model. The other mechanism is based on the late binding properties of Java.

4.1 Introduction

4.1.1 Change at run-time

A structural property of software seems to be that changes are needed from time to time. Even in the situation that we would be able to deliver systems without bugs, exactly meeting the specified requirements, the dynamic environment in which the piece of software struggles for life would eventually dictate new or different requirements, which can only be met by changing the software system.

In fact, a requirement of probably all software systems is a certain degree of flexibility with respect to other requirements. One should design for change.

In several systems, shutting the system down to apply changes is unacceptable. According to Stankovic, capabilities for change at run-time will especially be needed in the field of real-time and embedded systems [Stankovic 1996].

4.1.2 Software architecture

The recently emerged field of software architecture addresses the design of overall system structure. Design for change should start at this level.

Software architectures are typically described as a composition of high-level connected components [Garlan, Allen et al. 1994]. The expression has often been used to indicate structures representing the development view of a system, i.e. the high-level structure of the code (Witt, for instance, uses 'software architecture' in this way [Witt et al. 1993]).

In recent years, software architectures more and more describe the high-level design of the software system as it is seen during execution, with connections representing 'interacts' relationships as opposed to 'implements' relationships [Allen and Garlan 1994]. Figure 4.1 shows these two usages of software architecture, with 'run-time view' indicating the latter concept [Stuurman 1997].

Kruchten extends this concept of software architecture into four views: the logical view, supporting the functional requirements; the process view, focusing on concurrency and synchronization aspects; the physical view, mapping software on hardware; and the development view, describing the implements relationships [Kruchten 1995].

In this article, we will adhere to the notion of software architecture as components connected by 'interacts' relationships, which generally means a combination of the logical and process view as seen by Kruchten.

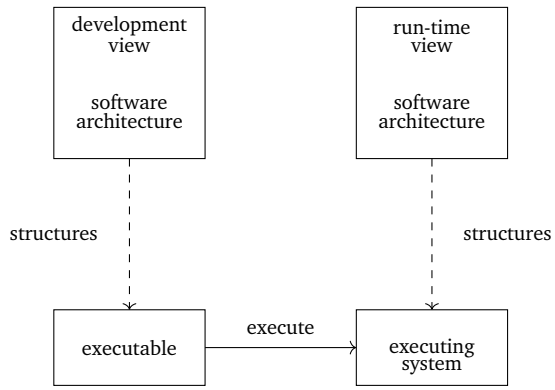


Figure 4.1: Development and run-time view software architectures

We will tackle the problem of change at run-time at the software architecture level, using a real-life example, and discuss the problem more generally.

4.1.3 A case study

The case study we use to experiment with our ideas is the control of unmanned vehicles on the 'Maasvlakte' in the ports of Rotterdam. Rotterdam is one of Europe's main ports, where huge numbers of containers are handled.

In the harbour area, containers are moved to and from ships, onto trains and lorries. Cranes are used to carry the containers from a ship onto a lorry, from a lorry to a stack, from a stack onto a train, or the other way around.

Because of the huge safety risks involved, the harbour currently employs a system of automated, unmanned vehicles for the transport of the containers. In the near future, extensions of this system are foreseen. More terminals will be involved, the number of vehicles will increase from the current number of 50 to 1000, and vehicles with different characteristics will be employed.

The current system, controlled from a central site, cannot handle these changes. The system we are designing should not only be usable for the current situation, but also for future changes, and should allow large scale simulations as well. Scalability and evolvability thus are important non-functional requirements of the system to be designed.

4.1.4 Outline of the article

The article is organized as follows: In Section 4.2 we give an outline of the system we designed, with communication between the processes according to the so-called 'subscription model'.

The two different mechanisms for change at run-time that are available in our solution for the control system are discussed in Section 4.3. One of the mechanisms is associated with the decoupling of processes as a result of the choice for subscription-based communication. The other mechanism is based on the late binding properties of Java. We show implications of both mechanisms on the software architecture.

Related work is discussed in Section 4.4. In Section 4.5 we discuss implications of what we have found, come up with advantages and disadvantages of both mechanisms for change at run-time, and show some loose ends. With Section 4.6, we end with concluding remarks, and express wishes for future research.

4.2 A control system for the Maasvlakte

The problem of the case study comes down to control the movements of unmanned vehicles from one given place to another: the problem of assigning tasks to vehicles is another subproblem.

In an early stage, we took two decisions:

- Control is distributed. Each vehicle has its own controller dictating its behavior. The main reason for this decision is that a distributed model conceptually fits perfectly to the situation of unmanned vehicles moving around. In practice, we will have to evaluate performance before deciding whether central or distributed control is more effective.
- Communication between the vehicle controllers and other processes is modeled according to the 'subscription model', which is explained below.

4.2.1 Subscription model communication

Subscription-based communication between processes [Boasson 1996] is a way of loosening coupling between processes. A conceptual model for subscription-based communication is the use of radiographic frequencies, called: 'channels':

- The availability of an unbound number of channels is assumed.
- For each channel, the type of data that can be handled is defined.
- Each process may send data using whichever channels it wants, as long as it adheres to the type of data that is expected.
- A process may subscribe to channels. Processes are able to listen to the data sent along the channels to which they are subscribed.
- Subscriptions may be done (and undone) dynamically.

An analogy to this form of communication is the way usenet users talk to each other through newsgroups. Someone may send messages to newsgroups. One subscribes to the newsgroups one is interested in, and reads messages appearing in

those groups. Users have no need for direct connections, and may, in principle, stay anonymous.

In a system of distributed processes communicating according to the subscription model, direct connections between processes are avoided. However, one must keep in mind the restrictions of this form of communication:

- The order in which the data are sent is not necessarily the same order at which they are received.
- There is no way to ensure that data are not lost, other then sending some precious data twice or more times. The typical use of subscription-based communication is where the same kind of data is sent over and over again, every time slightly different [Boasson 1998]. In such a case, the loss of one message is no disaster: the receiving processes temporarily use information slightly older than it should be.

4.2.2 Outline of the control system

The control system we designed for the Maasvlakte case study is not discussed in detail in this paper; we sketch the outline here.

The whole area where the vehicles may drive is characterized by x and y coordinates. Each unmanned vehicle is controlled by its own vehicle process.

We assume that:

- A seperate planner system provides each vehicle with a plan: the place where it should receive a container from a crane, and the place where it should deliver the container. Places are represented by their coordinates.
- A vehicle process is able to deduce a detailed plan (a sequence of places) from the given plan.
- Each vehicle process knows the position (in terms of coordinates), the velocity and the characteristics of the vehicle it controls.

Vehicles send their short-term plans (the part of the detailed plan that should be followed in the immediate future) regularly through a channel. They listen to the short-term plans of other vehicles and evaluate possible collisions. In the case of a possible collision, traffic rules determine which vehicle should wait for the other.

To avoid scalability problems, we divided the whole area in so-called regions: a grid of for instance 10 by 10 coordinated points. Each region is associated with one channel. Each vehicle sends its short-term plan through the channel associated with the region it is positioned in; it listens to the same region channel and to the channels of the eight regions surrounding this region.

Figure 4.2 shows two vehicles in two different regions. Each vehicle sends (solid arrow) to the channel associated with the region it is positioned in. Each vehicle listens (dashed arrow) to the same channel, and to the channels belonging to the

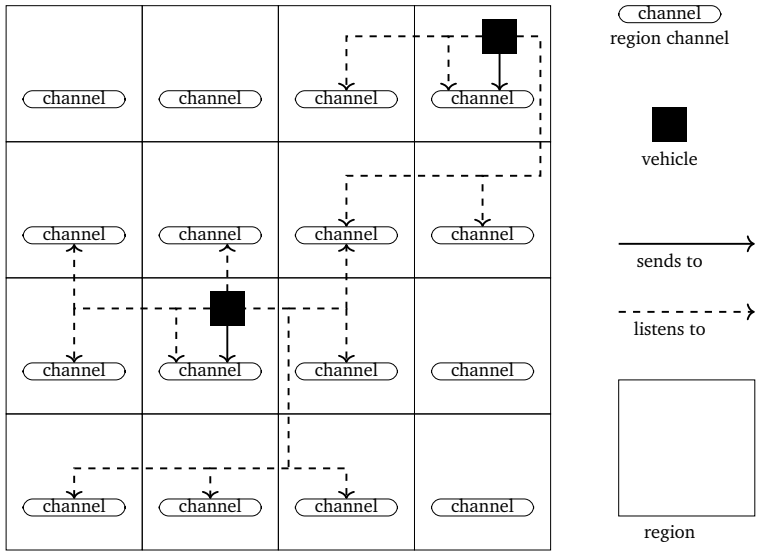


Figure 4.2: Vehicles in regions

eight regions around it. Note that it is possible that more than one vehicle drives within one region.

One region process is associated with each region (those processes are not included in the figure). It collects the data of the vehicles driving in the region and creates a summary which is sent through an 'image channel' (not in the figure), for the visualization system. The visualizer can also be used to zoom in to one region by telling it to listen to the associated channel. The short-term plans of the vehicles in that region may then be inspected.

Traffic rules are included in each vehicle process. Every vehicle process listens to a 'rules channel' (not in the figure) which is used to send a new version of the traffic rules. To avoid version conflicts, vehicles send the version number of the rules they use together with their short-term plan. In case of a version conflict, a default rule is used by both parties.

4.2.3 Software architecture large-scale

Figure 4.3 shows the software architecture of the proposed solution for the Maasvlakte case. A vehicle process (processes are represented by a circle) sends to a region channel, and listens to nine region channels (the region channel of its own region, and the channels of the eight regions around this region).

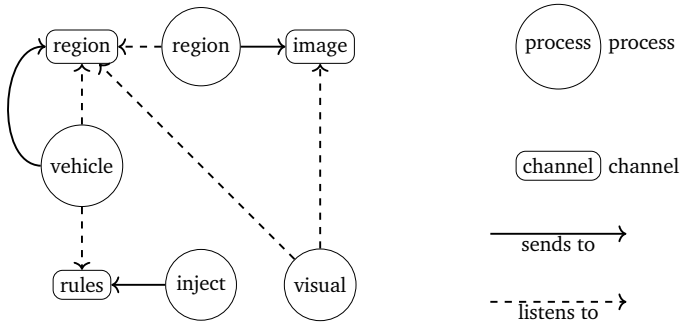


Figure 4.3: Configuration of processes and connections

A region process summarizes the data going through its region channel, and sends the summary to the image channel. The visualizer process may either listen to the image channel and receive the summaries of all regions, or listen to one region to receive detailed information. Every vehicle process listens to the rules channel which is, from time to time, provided with new rules by the rules injector.

4.2.4 Implementation

The system is implemented in Java. At this moment, we have a simulation of the system running on the computer network of our research group. We are currently evaluating the possibilities of a 'transport lab' with real miniature vehicles communicating through radio transmission.

Communication according to the subscription model is implemented in the 'channel library' developed within our research group [de Rooij 1998]. This library offers a set of classes with facilities to send and receive objects in the form of (unthreaded) `Transmitter` classes and (threaded) `Receiver` classes. One subclass of the `Receiver` class is meant for class definitions. It transforms a received class definition into a Java class and instantiates the class into an object. This class makes use of the late binding properties of Java. However, in Java, new class definitions are 'pulled' by the class needing it; in this case, we had to 'push' new class definitions into the system and force processes to instantiate them.

Figure 4.4 shows the internal view of a vehicle process. In this figure, active (threaded) objects are represented by a rounded box, while sharp-edged boxes denote unthreaded objects.

The vehicle object performs a loop:

- It asks the `VehicleState` object for the speed and position of the vehicle, the short-term plan and the current version of the traffic rules, and adds its identity to construct the current `VehicleInfo`.

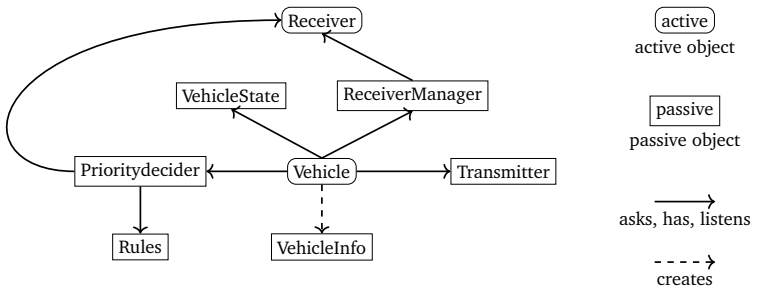


Figure 4.4: Objects in the vehicle process

- It asks the `Transmitter` to transmit the `VehicleInfo` object.
- In the meantime, the `Receiver` objects connected to the `Receivermanager` object listen for information from other vehicles. The `ReceiverManager` object is responsible for the choice of the channels to listen to. It selects these channels based on the position of the vehicle.
- Once in a while, the `PriorityDecider` is asked to compute the next position to drive to, based on the information gathered, and the traffic rules. The `VehicleState` object is updated.
- The traffic rules themselves may be updated when a new class definition is sent through the `Rules` channel.

4.3 Change at run-time at the Software Architecture Level

In the solution for the Maasvlakte case, the initiative for changes always lies outside the system (the initiative is taken by humans). Two mechanisms to apply these changes are available:

- create or remove processes
- send a new subclass.

4.3.1 Change at run-time at the process level

The reason that change at run-time in the form of the creation or removal of processes (or both) can be applied very easily in the system for the Maasvlakte, is the fact that we made use of the subscription model for communication.

Figure 4.5 shows a possible configuration of processes, channels, send- and listen connections. A solid-line connection stands for a sends-to connection; a dotted line

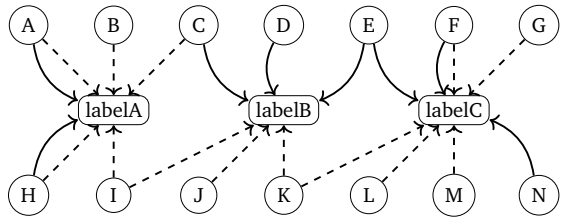


Figure 4.5: Subscription-based communication

for a listens-to connection. Processes never have a direct connection to other processes, and are anonymous. Channels are named, so processes need to know the name and the type of data of the channels to which they send or from which they receive data.

When a process is removed, no other process has to be informed. The same applies for the creation of a new process. Channels can be added too, without affecting other channels, or any of the existing processes.

What is needed however, to change a software system communicating according to the subscription model, is an 'all-knowing' entity, that should be added to the system. This 'all-knowing' entity has a direct connection to all processes, to be able to delete them. Moreover, it knows the names and the datatypes of every channel.

Making use of process creation and removal to induce change at run-time in a subscription-based system, means that one deviates from the model, and allows one process to have direct connections to every process and every connection within the system.

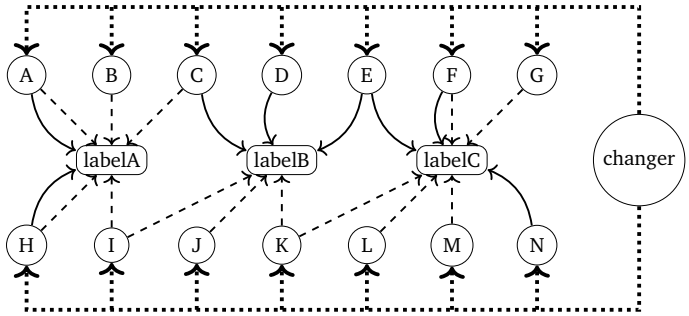


Figure 4.6: Subscription-based communication

Figure 4.6 shows this need for direct connections. The 'changer' in this figure may stand for a human, for a seperate process, or for one or more of the processes within the actual system. The type of connections needed for such a system are:

A sends-to connection: From process to channel. Belongs to the subscription model.

A listens-to connection: From process to channel. Belongs to the subscription model.

A create-or-remove connection: From process to process. Deviates from the subscription model.

A create connection: From process to channel. Belongs to the subscription model (for reasons of clarity, we did not include such a connection in Figure 4.6).

4.3.2 Change at run-time at the class level

Enforcing change at run-time by the mechanism of sending new classes is possible when every process that has to be changed listens to a channel reserved for the communication of class definitions.

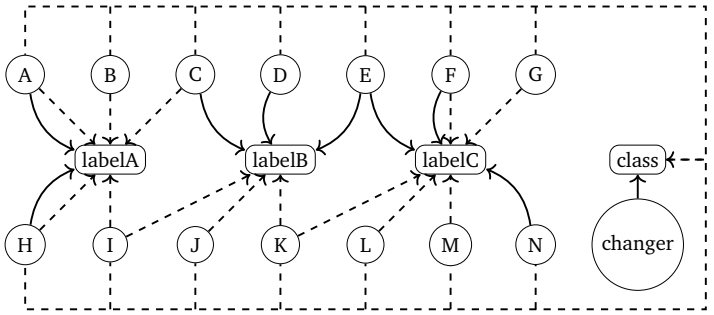


Figure 4.7: Sending new subclasses

Figure 4.7 shows that in this case, the connections available in the subscription model are sufficient.

Processes should all listen to a channel reserved for class definitions. Upon arrival, the class definition is instantiated into an object, which should connect itself to objects in the process. In the case study, this change mechanism is only used for the traffic rules, but in principle every class used in the processes of a system may be changed using this mechanism.

4.3.3 Initiating a change at run-time

As we have seen, an advantage of a subscription-based architecture is the ease of change at run-time. Both mechanisms are based on the fact that processes in this architecture are anonymous.

Changing the system at the process level means that one introduces a meta-level: one (or more) processes need direct connections to processes that are to be removed. In general, this 'meta-level system', will consist of a change management system and

the control system itself. The control system is wired according to the subscription model, while the change management system maintains direct connections to the processes of the control system.

When changing the system by sending a new class definition, the processes that are affected are anonymous too: processes that listen to the `Class` channel and that use the communicated class will change their behavior. The sender of the new class does not need to know which processes receive a new class definition.

This mechanism for change at run-time is more in line with the nature of distributed systems: there is no central control, not even for implementing changes. This feature opens the possibility of implementing systems that are able to change themselves, from within: one of the processes might notice the need for change of a certain class. This process, within the system, may send a class definition for such a new class, and thus induce a change in every process using that same class. Such a scenario is not possible when using change at run-time on the process-level, because the initiator of a change should know which processes are involved, and should have direct connections to those processes.

Change at run-time at the process level thus is most suitable for situations where the initiative for a change lies outside the actual system. A change management system is added to the control system, which can be used to check the validity of proposed changes, maintain consistency, etcetera. This change management system should maintain direct connections with the processes of the actual control system.

Changes at the process level may be used within the control system itself as well, but in this case, direct connections are needed within the control system, which is not obvious for a system based on anonymous communication.

Changes at the class level on the other hand, may be induced by every process within the control system, without any need for a change of the used subscription-based architecture. Of course, changes at the class level may be initiated from the outside as well.

4.4 Related work

A software architectural style which is closely related to the subscription-based model is the implicit invocation style [Garlan and Notkin 1991]. In this style, components communicate anonymously as well. The difference is that implicit invocation is a mechanism to call a method of another component anonymously (by raising an event, upon which all components that have subscribed to the event are called by their associated method). This means that on-line removal of components requires more attention than in the case of subscription-based systems: when a component is removed, all components that are calling a method, anonymously or not, are affected.

The Distributed Software Engineering Group of Kramer and Magee has done a lot of work on change at run-time in distributed systems. Kramer and Magee introduce a model for dynamic change management in distributed system [Kramer and Magee 1990]. They specify changes at the architectural level (without mentioning it explicitly: the paper was written before 'software architecture' became a topic). Changes

have the form of creation or removal of processes. The software architecture they implicitly assume is one with direct connections between the processes. A change management system is used to make certain that changes that have an effect on certain processes are only applied at moments when these processes are not involved in a communication transaction. This notion of a 'safe state' is elaborated upon by Goudarzi and Kramer [Goudarzi and Kramer 1996].

By using the subscription model, we avoid the problem of having to check for a safe state: because communication takes place through channels, no harm is done when a process is killed while sending or receiving data.

Magee and Kramer describe a specification language for software architectures, 'Darwin' [Magee and Kramer 1996]. Using Darwin, it is possible to specify the creation or removal of processes and the associated connections. Darwin might be a good candidate as an ADL for subscription-based software architectures.

Oreizy et al. define run-time architectural change in terms of components: they discern addition, removal and replacement of components, and reconfiguration [Oreizy et al. 1998]. The focus is on one architectural style, C2 [Taylor et al. 1996]. Tool-support for change at run-time in C2-based systems is proposed. Change at run-time at the component level is, again, associated with direct connections between components.

Segal and Frieder describe a scheme for procedure replacement [Segal and Frieder 1988]. Our class-level mechanism is, like this scheme, a change at source code level. We use the decoupling aspects of the software architecture, together with the late binding properties of Java, to enable the use of source code changes at the architectural level.

4.5 Discussion

Change at run-time, at the architectural level, is a rather new research area. The approaches that we have seen until now aim at the creation and removal of components in an architecture with direct connections.

Because we use a subscription-based model, the problem of having to apply changes at the right moment is non-existent. Another reason why we do not have to worry about lost messages is the nature of subscription-based communication. It should be used for information that is sent over and over again, every time slightly different. In such a case, it is no problem when some of the data are lost during the removal or the creation of a process.

On the other hand, we make use of the same connections for the distribution of class definitions. As has been said, the only way to handle precious information in the subscription model, is to send it more than once. Within the subscription model, there is no mechanism to ensure that eventually every process will receive a new class definition, and it is impossible to know whether or not every process did receive a new class definition.

This fact has implications on the internals of the processes involved: one must always take the possibility into account that two processes might have different versions of the same class. In our Maasvlakte case, where we send new traffic rules

in the form of class definitions, we handle this inconsistency problem by having the vehicle processes send the version of the traffic rules they use, together with their short-term plan and other information. In the case of a version conflict, every process involved uses one general default rule.

Change at run-time by creation and removal of processes is not new. New processes may get connections to existing channels, or new channels may be created. When using this kind of change at run-time in a subscription-based software architecture, one avoids connection-related problems on the one hand, but on the other hand, one needs to violate the software architecture to enable the removal of processes.

Change at run-time by the distribution of new class definitions is new. This mechanism is facilitated by the late binding properties of Java, but on the other hand, we had to find a way to enforce receiving processes to use a received class: normally, new versions of Java-classes are communicated on a pull-base, while we needed communication of class definition on a push-base.

In fact, 'precious' information should be sent through direct connections. This enables one to use protocols to check whether the information is really received, or to take action in the case of failing processes or failing connections.

A wish for future research is to explore combinations of direct connections and anonymous communication.

Changes at the class level can be implemented by removal and creation of a process, provided that one has the means to save the state of a process. The same applies for the other way around. So, the choice for process level or class level changes should be based on other considerations than the implementation. One such a consideration is the question wether the initiative for changes comes from within or from outside. Another consideration could be the fact that direct connections from outside the system might necessary anyway because the user wants to have direct control.

4.6 Conclusion

We have presented two mechanisms for change at run-time in a control system consisting of distributed processes, communicating according to the subscription model.

One mechanism is to remove and create processes. As a result of the used communication model, problems of finding a safe state do not exist. On the other hand, one needs direct connections to the processes involved.

The other mechanism makes optimal use of the anonymous connections, sending new class definitions through channels. The problem in this case is that the application should be able to handle version conflicts.

We already mentioned that a wish for future research is to explore the advantages and disadvantages of combining direct and anonymous connections.

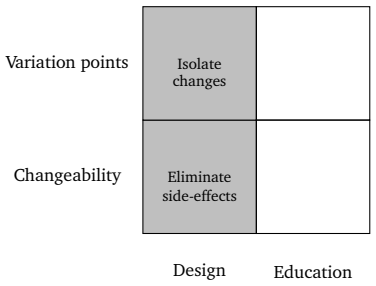
For the future, we would like to explore both mechanisms for change at run-time. We would like to build a change management system, and elaborate on the different kind of consistency checks that might be done by such a system.

With respect to change at run-time by distributing new class definitions, we would like to build ready-to-use components for control systems in the domain of logistic problems. With these components, one should be able to build distributed control systems, with the possibility to update the used classes while the system is running.

In order to build a system that enables users to construct a system by wiring together components, we need a formal description of the software architecture that we use. Darwin looks like a candidate specification language to do so.

5

Flexible feedback services for exercise assistants[1]



In this publication, several techniques to design for change have been applied. Both the software architecture and the generic nature of this framework for exercise assistants make it possible to use the framework for different domains.

A feedback engine that provides semantically rich feedback on steps toward a solution of an exercise in a formal domain such as logical expressions, has been developed using generic programming techniques in the functional language Haskell. This feedback engine is based on a framework for exercise assistants [Jeuring et al. 2007].

The description of the formal domain, the description of the exercises, and the description of the strategies that can be used to solve exercises, are separated from the remainder of the code. They are specified in an Embedded Domain Specific Language (EDSL). Adding another domain or another class of exercises (with their own strategies to solve exercises) is possible just by supplying a description of the domain and the strategies; there is no need to apply changes to the remainder of the code.

The back-end of the exercise assistants that can be derived from this framework, is decoupled from the front-end by offering the back-end in the form of a set of light-

[1]This article is based on two publications: 'Feedback Services for Exercise Assistants' in the Proceedings of the7th European Conference on e-Learning, pages 402-410, 2008, and on 'A Generic Framework for Developing Exercise Assistants' in the Proceedings of the 8th International Conference on Information Technology Based Higher Education and Training, ITHET, 2007

weight web services. Thus, various front-ends may use the same web services to create various exercise assistants.

The candidate was involved in creating one such a front-end and in testing an exercise assistant with students [Lodder, Passier et al. 2008]. Since the publication of this article, the feedback engine has been extended [Heeren and Jeuring 2014].

Relevance to this thesis

The framework described here is a beautiful example of the strategy to separate those parts that will need to change from the remainder of the code. It is also an example of the strategy to reduce side-effects, and to apply loose coupling in the form of web services.

Abstract

Immediate feedback has a positive effect on the performance of a student practising a procedural skill in exercises. Giving feedback to a number of students is labour-intensive, and as a result, providing immediate feedback is often hardly possible for a teacher. To alleviate this problem, many electronic exercise assistants have been developed. However, many exercise assistants have shortcomings with respect to the feedback they offer.

We have a framework that gives semantically rich feedback within several domains (such as logic, linear algebra, arithmetic), and can be extended to support new domains with relatively low effort. We need to have knowledge about the domain, about how to reason with that knowledge (that is: a set of rules), and we need a specified strategy to use those rules. We offer the following types of feedback: correct/incorrect statements, distance to the solution, rule-based feedback, buggy rules, and strategy feedback.

We offer the feedback functionality in the form of light-weight web services. These services are offered according to a protocol, for example, JSON-RPC. The framework is set up in such a way that it can be easily extended to support other protocols, such as SOAP. The services we provide are already used by existing exercise assistants, for instance, by MathDox, ACTIVEMATH, and by our own exercise assistant.

Our feedback services offer a wide variety of feedback functionality; exercise assistants using our services can construct different kinds of feedback. One possibility for instance, is to start giving correct/incorrect feedback, and only start to give semantically rich feedback on individual steps when a student structurally fails to give a correct answer. Another possibility is to force the student to take one step at a time, or to follow one specific strategy.

In this article, our main focus is on the description of the feedback services we offer. We briefly discuss the feedback engine that serves as a back-end of our feedback services. We will give examples of how to use our services, and we will show an example web-based application that uses the feedback services, for the domain of simple arithmetic.

5.1 Introduction

Receving feedback is essential for a student's learning process. In a classroom setting, feedback is given by a teacher. When a student goes astray, a teacher may give feedback to a student on how to get back on the right track. However, giving feedback to a number of students is labour-intensive.

Many electronic exercise assistants have been developed to support a teacher in this regard. Exercise assistants in many domains, such as logic, algebra, calculus, etcetera, support a student to practice skills. These exercise assistants usually offer a rich user interface and various kinds of feedback. Exercise assistants may support a large number of students.

Another advantage of an exercise assistant is that feedback may be given immediately, which is nearly impossible for a teacher when the number of students grows. Research has shown that during exercises, under certain circumstances, immediate feedback improves the performance of a student [Hattie and Timperley 2007; Mory 2003; Morrison et al. 1995]. In *Rules of the Mind*, Anderson discusses the effectiveness of feedback in intelligent tutoring systems and observes no positive effects in learning with deferred feedback, and even observes a decline in learning rate instead [Anderson 1993]. Erev et al. also claim that immediate feedback is often to be preferred [Erev et al. 2006].

Providing feedback is of fundamental importance for the acceptance and usability of exercise assistants.

There are many exercise assistants [Beeson 1998; Chaachoua et al. 2004; Cohen et al. 2003; Goguadze et al. 2005], but the feedback they offer is usually limited, or labourious to specify. Some exercise assistants are limited to one domain, whereas others are not able to generate exercises, so exercises have to be listed by hand. To our knowledge none of the exercise assistants available today can generate feedback on the strategy (or procedural) level.

We have a feedback engine that provides various types of feedback, listed in Section 5.2, and can derive sematically rich feedback within several domains, automatically. We have made our feedback functionality available to other exercise assistants. Exercise assistants can extend their functionality with the feedback we offer.

The contributions of this paper are:

- We describe the feedback services we offer in detail.
- We give an example of how an exercise assistant could incorporate our feedback services.

The document is structured as follows. Section 5.2 lists the types of feedback we provide. We describe the web services that are to be used to access our feedback functionality in Section 5.3. Section 5.4 describes the interface details and illustrates the use of our services with an example application. Finally we describe related and future work and draw our conclusions in Section 5.5.

5.2 Feedback

Ideally a tool gives various kinds of detailed feedback. For example, when a student rewrites $\frac{1}{2} + \frac{1}{3}$ into $\frac{3}{6} + 2$, our feedback engine will signal that there is a missing denominator.

If $\frac{1}{2} + \frac{1}{3}$ is rewritten into $\frac{2}{5}$, the feedback engine will signal that an error has been made when applying the rule of adding fractions: correct application of this definition would give $\frac{5}{6}$.

Finally, if the student rewrites $\frac{2}{5} + \frac{3}{5}$ into $\frac{4}{10} + \frac{3}{5}$, it will tell that the fractions already have a common denominator and according to the strategy, the numerators should be added.

The first kind of error is a syntax error, and we are able to detect them because good error-repairing parsers, which suggest corrections to formulas with syntax errors, exist [Swierstra and Alcocer 1999]. The second kind of error is a rewriting error: the student rewrites an expression using a so called 'buggy' rule. There are interesting techniques available to find the most likely error in the case that a student rewrites an expression incorrectly [Bouwers 2007]. The third kind of error is an error on the level of the procedural skill or strategy to solve this kind of exercise.

Only very few tools give feedback at intermediate steps, other than in the form of correct/incorrect answers. Although correct/incorrect feedback at intermediate steps is valuable, it is unfortunate that the full possibilities of e-learning tools are not used. The main reasons probably are that the support of detailed feedback for each exercise is very labourious and that providing a comprehensive set of possible bugs within a particular domain requires a lot of research [Hennecke 1999]. Also, automatically calculating feedback for a given exercise, strategy, and student input is very difficult.

The following list gives an overview of the possible types of feedback:

Correct-incorrect statements: This type of feedback, offered by most exercise assistants, tells whether or not a given solution is correct. An example of an exercise assistant offering this kind of feedback is WisWeb (Freudenthal Institute, Netherlands)[2].

Distance to solution: Another type of feedback indicates the number of steps to the final solution and tells whether or not a rewritten term is a step closer to the solution. Using this information, a progress bar can be constructed, indicating the number of steps a student still has to take in order to solve the exercise. An example of an exercise assistant offering this kind of feedback is APLUSIX[3].

Rule-based feedback: This type of feedback gives feedback on the level of rewrite rules. Examples of this type of feedback are a statement which rules are applicable, or a hint about how to apply a particular rule. This kind of feedback can be used by the student to find out what went wrong with his or her own application of the rule.

[2] http://www.fi.uu.nl/wisweb/
[3] http://aplusix.com

Another approach is to use a set of valid rewritings of a certain term. A non-valid rewriting, submitted by a student, is compared to this set of valid rewritings; the closest one is then used to ask the student whether he or she maybe meant to use the rewriting rule that results in that specific rewriting.

Buggy Rules: Feedback can also be provided by analysing mistakes that are often made by students, and distill so-called 'buggy rules'. In case of an incorrect rewriting, the rewriting can be matched against these buggy rules. Feedback based on buggy rules informs the student which mistake was made. Buggy rules are usually labour-intensive to specify. An example of the buggy rules approach can be found in AlgeBrain [Alpert et al. 1999]. This approach can be combined with the rule-based approach, as is done in the Slopert project [Zinn 2006]. An example of a buggy rule is to add the numerators and denominators when adding fractions, e.g. $\frac{1}{2} + \frac{2}{3} = \frac{3}{5}$.

Strategy Feedback: Many domains, such as logic, algebra, or calculus, require a student to learn strategies. A strategy, sometimes called a procedure, describes how basic steps may be combined to solve a problem. For example, at elementary school, students have to learn how to add fractions. A strategy for adding fractions is: 'First find a common denominator, transform the fractions to this denominator, add the numerators, and try to simplify the resulting fraction'. If a student does not start with determining a common denominator, feedback on the strategy level might be: 'Before adding fractions, they should have a common denominator'.

We developed a feedback engine [Jeuring 2007; Heeren, Jeuring et al. 2008; Heeren and Jeuring 2009] that automatically calculates the aforementioned feedback. In the next section we discuss how other exercise assistants can use our feedback functionality.

5.3 Feedback services

We offer our feedback functionality in the form of on-line web services. Web services are better maintainable and easier to deploy than, for instance, a library. Exercise assistants can use our services and provide their users the feedback types discussed in the previous section. With the diversity in types of feedback and the simple interface of the web services (discussed in Section 5.4, the user of our services is fully in charge of the use of our feedback and of the presentation of to the students.

Virtually no restrictions in the use of our feedback services exist. An example is to start giving correct/incorrect feedback, and only then give semantically rich feedback on individual steps, when a student structurally fails to give a correct answer. Another possibility is to choose to only give feedback after the final submission of a student, showing a trace of his or her steps to the solution.

We can use the input from students/users, via logs and statistics, to optimise our feedback engine, for instance by distilling common mistakes and thus extract buggy rules.

5.3.1 Domains

To be able to derive feedback automatically, our feedback engine needs a description of a domain (like logic or arithmetic), a set of domain-specific rules (such as the De Morgan rules for the logic domain), and a strategy (such as: rewrite until the expression is in disjunctive normal form). Rules are specified as a set of rewrite steps. Strategies are specified in an embedded domain-specific language in Haskell [Peyton Jones 2003].

Our approach to specify strategies for exercises is generic and not limited to a particular exercise domain. We have developed a strategy language in which we can specify strategies ('first do this', 'repeat this' etcetera) [Heeren and Jeuring 2009]. From this specification we can create something that looks like a context-free grammar. The sentences of this grammar are sequences of rewrite steps (applications of rules). We can check whether or not a student follows a strategy by parsing the sequence of transformation steps, and checking that the sequence of transformation steps is a prefix of a correct sentence of this context-free grammar.

Besides providing feedback, our feedback engine can generate exercises for every domain that can be described using a domain description. We use the domain description for a tool named QuickCheck [Claessen and Hughes 2011], an automatic testing tool for Haskell, with which our feedback engine is able to generate random exercises. In the near future, we would like to automate this process, by making use of generic programming [Backhouse et al. 1999]. The difficulty of an exercise is may be specified as a parameter.

Our feedback engine is equipped with several domains, and we keep adding more domains. At the time of writing our feedback engine offers the following domains (and exercises):

propositional logic: bring a proposition in disjunctive normal form (DNF).

lineair algebra: Gram-Schmidt, solving a lineair system, Gaussian elimination, solving a lineair system with matrices.

arithmetic (including functions): simplifying arithmetic expressions.

relation algebra: bring an expression in conjunctive normal form.

Let us consider an example from the arithmetic domain: the problem of adding two fractions, for example, $\frac{1}{2} + \frac{1}{3}$.

The set of rules consist of:

add: $(\frac{a}{c} + \frac{b}{c} = \frac{a+b}{c})$

multiply: $(\frac{a}{b} \times \frac{c}{d} = \frac{a \times c}{b \times d})$

rename: $(\frac{a}{b} = \frac{a \times c}{b \times c})$

a buggy rule: $(\frac{a}{b} + \frac{c}{d} = \frac{a+c}{b+d})$

A possible strategy to solve this type of exercise is the following: find the least common denominator (LCD) of the fractions, rename the fractions such that LCD is the denominator, add the fractions by adding the numerators. Here, we show the various types of feedback we may encounter when a student tries to solve the exercise.

Correct-incorrect statements: If the student submits $\frac{5}{6}$ as the final answer, the feedback engine will indicate that the exercise has finished. If the student submits a wrong answer, and no buggy rule can be recognized, our feedback engine indicates that the submission was incorrect. Another possibility is that the following expression is submitted: $\frac{3}{6} + \frac{2}{6}$. Our feedback engine will recognise that the final solution has not been reached and will report that another step can be made. It can even tell which step the student has to take next.

Distance to the solution: When the student starts solving the exercise, the feedback engine can indicate the (minimum) number of steps it takes to solve the exercise. An exercise assistant can use this information to display a progress bar.

This information is useful in another way as well. Suppose that a student has rewritten the fractions in such a way that they have a common denominator (applied the rename rule). Instead of taking the next logical step (adding the numerators), the student now renames the fractions again. Although this rewriting may be valid, it is not the most efficient way to solve the exercise. The rewriting does not shorten the distance to the solution. An exercise assistant may convert this information into an appropriate feedback message.

Rule-based feedback: Whenever a student gets stuck during an exercise, the feedback engine may help. The feedback engine knows where a student resides in the strategy and can deduce the applicable rule(s) from the strategy. This information can be returned to the student in the form of a hint, for instance, 'Try to apply the rename rule'. If a student still remains stuck, the feedback engine can apply the rule itself, and show the student the resulting rewriting.

Buggy Rules: If a student makes an error, the feedback engine checks if a buggy rule (a common mistake) can be matched. If that is the case, the buggy rule is reported to the exercise assistant. An example of a buggy rule is to add the numerators and denominators($\frac{1}{2} + \frac{1}{3} = \frac{2}{5}$). An exercise assistant can tell the student what error he or she has made, and give a hint, using the rule-based feedback, about which rule he or she should have applied.

Strategy feedback: The feedback given in the previous examples often is already related to a strategy. A strategy defines the route towards the result. If a student deviates from the route, there are two possibilities: the student has made an error, and the feedback engine gives appropriate feedback, or the student performed a valid rewriting that is not part of the strategy. This is what we call a detour; it is up to the exercise assistant to decide which action to take, either to force a student to remain within the strategy, or to let him or her carry on, along a less efficient or unknown route.

Our feedback engine can be easily extended with additional domains. Instantiation of a particular domain is a labour-intensive process, but less labour-intensive than having to specify feedback by hand. Once the domain, rules and strategies are specified, we can calculate feedback automatically. It is no longer necessary to construct feedback by hand, which is an error-prone process. Our feedback services even include feedback on the strategy level, which is, to our knowledge, unprecedented.

In this subsection, we described the feedback that our services provide; the next subsection describes how the services themselves.

5.3.2 Web services

Our feedback services can be accessed on-line, through so-called web services. Web services support machine-to-machine interaction over a network [Haas and Brown 2004]. Web services have an interface description in a machine-processable format.

A network protocol is a set of formal rules that describe how data may be transmitted across a network. Currently, we support three protocols: JSON-RPC, XML-RPC (JavaScript Object Notation and eXtensible Markup Language – Remote Procedure Call), and a custom protocol for the MathDox project [Cohen et al. 2003]. The feedback service application has a modular architecture and can be easily extended with other protocols, for instance, SOAP (Simple Object Access Protocol). In this document we use JSON-RPC in the examples. The same examples using the other protocols are available at our website[4].

JSON is a data interchange format; it is a lightweight format and is easy to read and write for humans. At the same time, it is easy to parse and generate for machines . JSON is built on two structures: a collection of name/value pairs and an ordered list of values. JSON-RPC can be used to access our feedback services. It is a remote procedure call protocol encoded in JSON. It is a very simple protocol, with very few data types and commands. One may invoke our feedback services in JSON through a CGI (Common Gateway Interface) binary over HTTP. Here, we show an example with indentation to show the structure of a request URL; the service itself will be explained in Section 5.4.

```
http://ideas.cs.uu.nl/cgi-bin/service.cgi?input= {
  "method" : "allfirsts",
  "params" : [["Simplifying fractions", "[]", "1/2+1/3", ""]],
  "id" : 42
}
```

Note that the URL needs to be escaped from illegal characters (such as spaces and curly braces), but for clarity reasons we use a representation without escape characters. The CGI binary has one parameter called `input`. The example service call results in the following result reply:

```
{
  "result": [
    "Rename", "[]",
    [
```

[4] http://ideas.cs.uu.nl/trac

```
      "Simplifying fractions",
      "[0, 2, 2, 1, 0, 1]",
      "((3/6) + (2/6))", "[];"
    ]
  ],
  "error": null,
  "id": 42
}
```

In the next section, we explain the syntax and semantics of the service call and its result in more detail. We will also show how to embed the service calls through JSON-RPC in a web application.

An interesting feature of the protocol is that it is stateless. When necessary, the state is passed through in the form of a parameter. We represent the state as a four-tuple, containing (we append the values from the previous result of the example between parentheses):

- an *exercise identifier* ('Simplifying fractions'); an overview of exercise identifiers can be found on our website;
- a *location* parameter that holds the place in a strategy. A strategy can be expressed as a sequence of rewritings. The location parameter is encoded as a list of integers. The integers in the list encode which rewritings of the possible sets of rewritings have been used. If these rewritings form, together, a prefix of a complete strategy, we are in the middle of a derivation to the end solution, in line with a certain strategy. The location is such a prefix of a strategy: $([0,2,2,1,0,1])$;
- the current *expression* $(((3/6)+(2/6)))$;
- an optional *context* parameter, denoting the part of the expression we are working on ([];).

Because of the fact that the protocol is stateless, the state parameter should be saved when someone would like to continue the exercise at a later point in time. This mechanism offers exercises assistants the opportunity to save a student's work.

5.4 Service specification

In this section, we describe the semantics of every service along with its input parameters and its output. The name of a service is, at the same time, a hyperlink to the actual service. Our services can be reached through the following URL:

```
http://ideas.cs.uu.nl/cgi-bin/service.cgi?input=json_input
```

The general structure of the JSON input parameter, which should be supplied in the URL, is:

```
{
  "method" : <service name>,
  "params" : <list of parameters>,
  "id" : <id of the exercise>
}
```

What follows is a description of the services we provide. For some services, we describe the input and output in detail; the other services have a similar structure.

generate: The generate service, as its name indicates, generates a new exercise. It has two parameters: a string representing the exercise identifier and an integer value representing the difficulty of the exercise:

```
{
  "method" : "generate",
  "params" : ["Simplifying fractions", 5],
  "id" : 42
}
```

The result of a generate service call is a new state, containing a fresh exercise:

```
{
  "result": ["Simplifying fractions", "[]", "2/3 + 4/5", "[];" ],
  "error" : null,
  "id" : 42
}
```

In addition to the new state, the result value contains an error value, indicating whether there were any errors, and an identifier value.

derivation: The *derivation* service takes a state parameter as input:

```
{
  "method" : "derivation",
  "params" : [["Simplifying fractions", "[]", "1/2 + 1/3", "[];"]],
  "id" : 42
}
```

It returns a path to the end solution, in the form of a list of three tuples, containing the rule identifier, the location and the resulting expression:

```
{
  "result": [
    ["Rename",[], "((3/6) + (2/6))"],
    ["Add",[], "((3 + 2)/6)"],
    ["AddConst",[0], "(5/6)"]
  ],
  "error" : null,
  "id" : 42
}
```

allfirsts: The *allfirsts* service takes a state value as a single input parameter, and returns a list of three arrays, containing a rule identifier, a location and a new state value that would be the result of applying the rule. The list represents all valid steps a student might take from this location in the strategy. The head of the list is a rule that leads to a path to the end solution.

onefirst: The *onefirst* service is just the head of the list returned by allfirsts. This service is redundant; it is in the interface for convenience reasons.

applicable: The service *applicable* takes a location and a state as input parameter, and returns a list of all rule identifiers that are applicable to the current expression. The list also includes rules that are not within the strategy.

apply: The service *apply* takes a rule identifier, a location and a state as input, applies the given rule to the current expression, and returns a new state.

ready: The service *ready* determines whether an expression in a state input parameter, is the final solution. This service returns a boolean value.

stepsremaining: The service *stepsremaining* returns an integer denoting the number of steps towards the end solution, given the state.

submit: The service *submit* takes the current state and the student input in the form of an expression:

```
{
  "method" : "submit",
  "params" : [
    ["Simplifying fractions", "[]", "(1/2) + (1/3)", ""],
    "(3/6) + (2/6)"
  ],
  "id" : 42
}
```

It returns one of the following result codes, together with values, depending on the result code:

SyntaxError: A syntax error occurred.

Buggy: One or more buggy rules could be matched; a list with their identifiers is returned.

NotEquivalent: The student has made a mistake.

Ok: The submitted expression is equivalent, and one or more rules have been applied that are in line with the strategy. A list of applied rule identifiers and a new state are returned.

Detour: The submitted expression is equivalent, but one or more of the applied rules do not correspond to the strategy. A list of rule identifiers and a new state are returned.

Unknown: The submitted expression is equivalent, but none of the known rules matches.

In JSON, the result reply has the following form:

```
{
  "result": {
    "result": "Ok",
    "rules": ["Rename"],
    "state": [
```

```
      "Simplifying fractions",
      "[0, 2, 2, 1, 0, 1]",
      "3/6 + 2/6", "[];"
    ]
  },
  "error": null,
  "id": 42
}
```

Every type of feedback that we described out in Section 5.2 can be constructed using the set of services introduced in this section. For example, the 'distance to the solution' type of feedback is enabled by the *stepsremaining* service. The 'buggy rules' type of feedback is delivered by the *submit* service.

5.4.1 Client example

In this subsection we describe how to embed one of our services in a web application. Although we focus on a web application, the general idea can be used in any platform supporting remote procedure calls.

We use AJAX (Asynchronous JavaScript and XML) to give the example web application the responsiveness of a desktop application. Using AJAX, we call a service using an XMLHttp request. The parameters of the service call are in JSON. The response of a service call is in JSON as well; the exact structure of a response value has been explained in the previous subsection. The result values of a service call should be used to update appropriate areas of the web application. Thanks to AJAX, this can be done without a page refresh, resulting in a more responsive web application.

Many service calls return a new state value. The state value needs to be stored in order to keep track of the user's progress. How the state value is stored, is entirely up to the exercise assistant. It could, for example, be stored in memory or in a cookie.

The next piece of JavaScript code shows a service call:

```
function genExercise() {
  var exercise = jQuery("exercise");
  var myAjax = new Ajax.Request(
    "http://ideas.cs.uu.nl/cgi-bin/service.cgi",
    {
      method : "post",
      parameters : {
        "input" : {
          "method" : "generate",
          "params" : ["Simpifying fractions", 5],
          "id" : 42}
        },
      onSuccess : function(response) {
        var res = response.responseText.parseJson();
        exercise.html(res.result);
      }
    }
  );
}
```

Note that this code uses the well known jQuery JavaScript library, which provides a clean interface to make AJAX requests. The function `genExercise` declares a vari-

able `exercise` that points to an HTML element with identifier 'exercise'. A successful service call updates the HTML element with the result value.

The following HTML fragment shows how to display a button that invokes the function `genExercise`, and a field, with identifier 'exercise', which will contain the result value after a service call.

```
<html>
  <body>
    ...
    <div align="center">
      <button id="generate">Generate exercise</button>
      <h2>Generated exercise:</h2>
      <pre id="exercise">...</pre>
    </div>
  </body>
</html>
```

This small example shows that semantic rich feedback can be obtained, in a client, with a relatively small effort. The expressions in this HTML example are encoded in plain ASCII. To improve the usability, a better representation can be given by also using non-ASCII characters. However, these improvements are domain-specific and are not trivial to automate.

5.5 Conclusions, related work, future work

This paper introduces our feedback services that can be used by exercise assistants, to provide semantically rich feedback. The services include several types of feedback, inlcusing feedback with respect to strategy. We described the interface of the feedback services and gave examples of how to call our services and how to embed our services in a web application.

Our services are already being used by some exercise assistants: MathDox and our own web application [Lodder, Jeuring et al. 2006; Passier and Jeuring 2006]. We are working on a connection between our services and the ActiveMath tool.

We have set up a Wiki and issue tracking system (Trac) for the development of our feedback software and services. This system also provides access to our software repository, through an on-line source code browser. It can be reached by the following URL:

```
http://ideas.cs.uu.nl/trac
```

Related work: There are many exercise assistants that offer students an environment in which they can practise skills by solving exercises, such as MathDox [Cohen et al. 2003], Aplusix [Chaachoua et al. 2004], the Autotool project [Rahn and Waldmann 2002], ActiveMath [Goguadze et al. 2005] and MathPert [Beeson 1998], to mention just a few. They usually offer a rich user interface and immediate feedback to the student. However, most of them are limited to correct/incorrect feedback, because it is often difficult and labourious in these systems to catch and analyse mistakes.

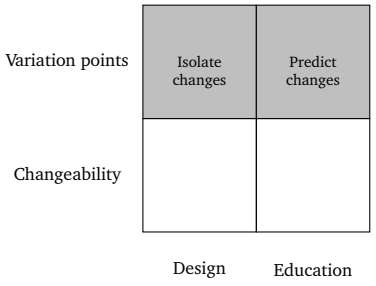
The ActiveMath project also provides services [Libbrecht and Winterstein 2005]. These services operate on a different level, and offer functionality to create a whole course. Our services focus at the level of exercises.

Future work: We continue our research and try to improve our feedback engine and services, by adding more domains and more protocols, to increase the number of exercise assistants that can use our feedback services.

A topic that needs further research is how to allow others, for example teachers or domain experts, to specify exercises, strategies and feedback messages.

6

Beautiful JavaScript: How to guide students to create good and elegant code[1]



The focus of this article is on educational aspects: on procedural guidelines that lead, when followed, to elegant code. One of the reasons that the resulting code in the example that we provide is resilient with respect to change, is the fact that changes are declared outside the software, in HTML. The article thus illustrates one of the strategies for change, and addresses the question how to teach students design for change as well. One of the guidelines is to imagine probable changes in the future, and refactor the program in such a way that easy implementation of those changes is addressed. The resulting code is more abstract.

We validate, in this article, that our guidelines, when applied, indeed can lead to 'elegant' code. With 'elegant' code, we mean code that adheres to the SWEBOK design principles that are introduced in this article. Code that adheres to these principles often is called 'elegant' or 'beautiful' [Oram and Wilson 2007]. What clearly needs to be done, is to validate the proposed approach with students, to see whether our guidelines in practise *do* lead to elegant code.

[1]This article originally appeared in the Proceedings of the Computer Science Education Research Conference CSERC, 2014.

Relevance to this thesis

This article addresses the educational aspect of design for change. It shows that design for change may help in deriving elegant, flexible code. In particular, the strategy to declare changes outside the code gets attention.

Deviations from the original article

We changed the order of the sections: Section 6.2 now follows the introduction. In both the introduction and Section 6.2, we added references on the (poor) performance of students on programming, and on related work.

Abstract

Programming is a complex task, which should be taught using authentic exercises, with supportive information and procedural information. Within the field of Computer Science, there are few examples of procedural information that guide students in how to proceed while solving a problem. We developed such guidelines for programming tasks in JavaScript, for students who have already learned to program using an object-oriented language.

Teaching JavaScript in an academic setting has advantages and disadvantages. The disadvantages are that the language is interpreted, so there is no compiler to check for type errors, and that the language allows many 'awful' constructs. One of the advantage is that, because of those disadvantages, programmers should consciously apply rules for 'good' programs, instead of being able to rely on the errors and warnings that a compiler will raise. Another advantage is, for instance, that it is very easy te create a user interface, because the user interface is declared in HTML, or that a program can be run directly, without first having to compile the code.

In this article, we show how we guide students to develop elegant code in JavaScript, by giving them a set of guidelines, and by advising a process of repeated refactoring until a program fulfills all requirements. To show that these guidelines can indeed lead to flexible code, we describe the development of a generic module for client-side form validation. The process followed and the resulting module both are valuable in an educational setting. As an example, it explains precisely to students how such a module can be developed by following our guidelines, step by step.

6.1 Introduction

Unfortunately, students are often not able to program after having followed introductionary programming courses. For instance, a great majority of these students are not able to perform the following steps: 1. Abstract the problem from its description, 2. Generate sub-problems, 3. Transform sub-problems into sub-solutions, 4. Re-compose the sub-solutions into a working program, and 5. Evaluate and iterate. [McCracken et al. 2001]. Another study reveals that students are, in general, not able to 'provide a summary of what the code does in terms of the purpose of the

code' after an introductionary programming course [Whalley et al. 2006]. Another example is the fact that only 17% of the students at the end of a first-year Java programming course held a viable mental model of a reference assignment [Ma et al. 2007]. Finding programming tasks too difficult is one of the major causes of dropout during introductionary courses [Kinnunen and Malmi 2006]. Finding these tasks too difficult might be related to a lack of procedural guidance. For example, one of the interviewed dropped-out students says: "It was so laborious to do the programming project considering the knowledge that had been given so far." [Kinnunen and Malmi 2006].

Learning how to program, according to general principles, can be seen as a complex task. Learning complex tasks should be based on real-life, authentic tasks [Merrill 2002]. Students should be provided with support and guidance while solving such a task. That guidance should tell students how to recognize an acceptable solution and should provide guidance to the solution process: procedural information is required when one offers authentic tasks to enable complex learning [Kirschner et al. 2006]. This means that one should provide procedural information when one teaches students how to program. Procedural information specifies how to perform the routine aspects of learning tasks, and, preferably, takes the form of direct, step-by-step instruction [van Merriënboer and Kirschner 2013].

Offering procedural guidance, in the form of a step by step approach to problem solving, is not very common in Computer Science education, as far as we know. We think that courses on programming could benefit from step by step approaches.

We developed a set of guidelines to create programs in JavaScript. The programming language Javascript has been developed within a short period of time, largely out of sight of academia. Javascript does not encourage encapsulation or structured programming, but strives to maximize flexibility, which may be a consequence of the fact that it was designed to allow non-programmers to extend web pages with logic [Richards et al. 2010]. As a result, Javascript has a number of 'awful' and 'bad' parts [Crockford 2008]. When using JavaScript, it is therefore very easy to create unstructured programs that do not satisfy general design and programming principles; programs with, as a consequence, a low level of reusability, understandability or adaptability.

On the other hand, Javascript has many 'good' parts too [Crockford 2008] and it is possible to create elegant code in JavaScript, adhering to design principles. Precisely the lack of language concepts such as modularity or information hiding makes Javascript a suitable language for learning how to program according to those principles. Students will have to force themselves to program in a clean way, according to programming principles, instead of being forced to do so by the language and/or the compiler.

Because the language does not enforce encapsulation and does not have easy solutions for private members or interfaces, students are forced to consciously apply those concepts to their programs, and as a result will become more aware of the value of the principles of, for instance, separation of concerns, information hiding, abstraction or modularity. As a consequence, students will get a more profound understanding of these principles.

By using our guidelines, students should be able to write elegant code. By 'elegant code', we mean code that adheres to the design principles that we will introduce later in this article.

As a first step in the validation of our guidelines, we have followed them ourselves, creating a module for form validation. We show that one can start with a program that is correct but does not comply with the software engineering design principles, and derive, step by step, code that is easily extendible, and conforms to the design principles of software engineering.

In this article, we describe the rules we give our students to recognize an acceptable solution, and we show the guidance for the process toward a solution. We will show how adhering to these guidelines leads to elegant code, by presenting a generic module for form validation as example. Even though there are several existing libraries for form validation, form validation is a fruitful subject when teaching students to apply design principles as abstraction, modularity, extendability and information hiding to JavaScript code.

6.1.1 Contributions

We offer a set of guidelines to derive programming code that adheres to the design principles of software engineering. We show how these guidelines may lead to elegant code by applying them to an example problem: client-side form validation in JavaScript.

Some of the guidance is specific for form validation or for JavaScript, but most of the guidance is applicable to other programming languages.

The resulting module and the process followed form an excellent illustration of how to follow our guidelines, and of the fact that these guidelines may indeed lead to elegant code. What needs to be done, however, is to validate whether the guidelines really help students in practice.

6.1.2 The structure of the article

In Section 6.2, we discuss related work. The following sections (6.3, 6.4 and 6.5) are structured according to the components that an environment for complex learning should have [van Merriënboer et al. 2002]:

Task description: A description of the task itself, which should be as authentic as possible. In this case, this is the task to create a module for client-side form validation. In Section 6.3, we describe the general requirements of a form validation module as a task description.

Supportive information: The information may be related to the domain of the task, or describes how students may recognize that a solution is acceptable. In Section 6.4, we describe both the specific rules to which a solution for a form validation module must adhere, and general design principles that must hold for any program.

Procedural information: Information to how students may proceed, which steps they should take while trying to solve the problem. In Section 6.5 we introduce the procedural information, in the form of a set of step by step guidelines. We show guidelines that are specific for this domain, as well as guidelines that apply to any program.

Part-task practice: Exercises that help students to reach a higher level of automation. We will not address these (smaller) exercises in this article.

In Section 6.6, we show that following our guidelines indeed leads to code that adheres to the design principles of Section 6.4. Finally, in Section 6.7, we discuss our work, draw our conclusions and define future work.

6.2 Related work

Form validation using a specification declared in the HTML is not new. For instance, Powerforms [Brabrand et al. 2000] specifies additions to HTML that are parsed by JavaScript. There are more libraries for client-side form validation, such as the jQuery Validation plugin. Here, form validation is reduced to specifying which fields have to be validated against which rules and, if a rule is broken, which feedback message should be presented. There are about fifteen different validation functions to choose from (for instance, email, creditcard or url). Our feedback library, while much barer, is simpler with respect to validating functions: all information is declared in the HTML, using standard attributes. However, our focus is not on the library itself. We have tried to show that the guidelines we give our students lead to elegant code (code adhering to general design principles), even if the first attempt is far from elegant.

One example of procedural guidelines on programming are detailed guidelines for functional programming [Felleisen 2001]. We use similar guidelines for creating functions; we have not shown these guidelines here. The guidelines we present here are meant for JavaScript, but can be easily adapted to support other programming languages.

Another example of procedural guidelines for programming is the STREAM framework [Caspersen and Kolling 2009], a step by step approach intended for novices learning OO-programming. The approach is based on stepwise improvement consisting of extension (extending the specification to cover more use cases), refinement (refining abstract code to executable code), and restructuring (improving non-functional aspects of a solution without altering its observable behavior). The arguments for the necessity of procedural guidelines for students that the authors give are in line with our own observations. Our approach is geared toward more advanced programmers. Our guidelines are less specific, and we offer students a means to check their code with the principles that the code should adhere to when good enough. Students who did learn to program using the STREAM method, could learn advanced programming using (an adaptation of) our guidelines.

An approach called 'goals and plans' in a visual programming environment, where a goal is 'a certain objective that a program must achieve in order to solve a problem',

and a plan is 'a fragment of code that performs actions to achieve a goal.', has a positive influence on the students' understanding of programs [Hu et al. 2012]. This approach offers students a form of procedural knowledge. Compared to our guidelines, the procedural knowledge is less detailed, and it is geared to novice programmers, while our guidelines are for more advanced students. Another study showed that providing students with explicit 'strategies' to use goals and plans resulted in better grades [De Raadt et al. 2009].

As far as we know, there are no other studies than ours on the role of procedural information for the complex task of programming web applications and event-driven programming or for advanced programmers.

6.3 Task description: requirements

6.3.1 Validation of input data

Form validation is the process of comparing data entered in a form against specifications for these data. Common validation procedures include checking for missing data (such as missing a social security number), invalid data (such as an impossible zip-code), and inconsistent data (such as a not existing combination of address and zip code) [Oliveira et al. 2005]. Some of these validation checks are obvious, such as a name - password combination. Other checks require more complex processing, such as checking the existence of a combination of address and zip code.

Validation of input data is needed both to keep the data stored on the server accurate and for security reasons, for instance, to prevent integer overflow and SQL injections [Pietraszek and Berghe 2006]. This means that the server thus must always validate any incoming data before storing them.

For usability reasons, however, it is preferable to validate the data that are entered by the user at the client as well. If not, a user would have to fill out a form, submit it, and wait for the response page that might tell the user that some of the entries are missing or are invalid. Experiments suggest that feedback should be given immediately after a user submits a form (and not after having entered a single field because users appear to get confused when that happens), and should be embedded in the form [Bargas-Avila et al. 2010].

Specifications for form entries should adhere to the robustness principle [Postel 1981], which means that one should not be too strict on the format that the user should adhere to, and translate the entered data into a format that the server expects. It should be possible, for instance, to enter a zip code both with and without a space between the numbers and the letters, and it should be possible to enter the letters both in upper- and lowercase.

The four dimensions of intrinsic information quality are authorization of the person who entered the data, timeliness of data entry, correctness, and completeness of the data entered [Ballou and Pazer 1985]. Authorization can only be checked on the server, and the same applies for timeliness. For client-side form validation, therefore, the focus is on completeness and correctness. Completeness means that all required data is entered, which means that no required fields are empty; correctness means

that all data are correct with respect to their specification. Correctness only needs to be checked if a field is complete (either has a value or is optional).

Browsers that support form features of HTML5 already validate user input according to the type attribute (such as date). There are even some browsers that validate user input according to the value of the pattern attribute. In these browsers, validation using JavaScript is not really necessary, but it is still advisable, because feedback given by browsers is often confusing. Furthermore, browsers are not able to validate related input fields such as a street name, a house number, a city name, and zip code. Client-side form validation in JavaScript thus is still necessary, and will be necessary at least in the near future.

6.3.2 Requirements for the solution

As we have seen, client-side form validation should focus on completeness and correctness. Completeness of an input field means that the field, if required, is not empty. Correctness of an input field means checking one or more of the following properties:

- The text has the right *format*. Form data are received, by the script, in the form of a string. Such a string should be parsed to check whether the input has the expected format (according to the robustness principle). For example, a (Dutch) zip code is formed by a sequence of four numbers, an optional space, and two letters.
- Tha value is within a specific *range* defined by a minimal and maximal value. For example, the age of a person should be between eighteen and hundred and twenty.
- The *combination* of data is correct. For example, the street name, house number, zip code and city name entered should be a valid combination.

Combinations of type and range will often occur, for example, an integer within a certain range or a string value out of an enumeration of strings. A variation of the second property is to require only a minimum or maximum value (for example, a person should be 18 years or older). Of course, there is always a default minimum and maximum: the minimum and maximum supported by JavaScript.

The third property (combination) is often validated on the server, because one needs to look up data in a database, which will, in general, not be available at the client-side. Therefore, the third property falls outside the scope of this task.

6.4 Supportive information

There are general design principles and programming principles that apply to all software. For the domain of form validation, there are some specific design principles. These design principles are a form of supportive information, because they tell the

student how to check whether a solution is satisfactory. Of course, general knowledge on JavaScript and HTML also falls in the category of supportive information, but for the purpose of this paper, we focus on the design principles.

6.4.1 General design principles

The Software Engineering Body of Knowledge [Bourque and Fairly 2014] contains the following software design principles:

Abstraction: Abstraction is 'a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information' [Allen, Barnum et al. 2009] and may consist of removing detail (to simplify and to focus attention) and of generalizing, and identifying the common core or essence [Kramer 2007]. Abstraction can be used as a means to decrease complexity or to enhance reuseability.

Coupling and cohesion: Coupling is the interdependence among modules, classes or functions (which should be minimized), and cohesion is the strength of association of the elements within a module, class or function (which should be maximized) [Allen, Barnum et al. 2009].

Decomposition and modularization: Software should be divided into components with well-defined interfaces. Decomposition and modularization is usually accompanied by separation of concerns, placing different responsibilities in different components.

Encapsulation/information hiding: Internal details of an abstraction are hidden, not available to external components.

Separation of interface and implementation: This can be seen as a form of information hiding: components offer a public interface, and hide their implementation.

Sufficiency, completeness and primitiveness: A software component is sufficient and complete if it captures the important characteristics of the abstraction it implements and nothing more. Primitiveness means that simple is better than complicated.

Separation of concerns: The notion of separation of concerns has been coined by Dijkstra [Dijkstra 1982]. A concern in software architecture is an interest of one or more of the stakeholders of a system, but in the expression 'separation of concerns', a concern is an aspect of the system that is being designed. Separating concerns is a means of handling complexity.

6.4.2 Specific design principles for form validation

For form validation in a web application, there are some specific, generally accepted, design principles:

Server-side validation is mandatory: All data must be validated on the server, because client-side validation can easily be bypassed and/or a user can have switched off the Javascript engine. The server is responsible for security, and for guarding the integrity of the stored data. Client-side form validation has a different purpose: usability, and user experience.

Enough is enough: Only those validation functions should be performed that 1) do not need secure information from the server, and 2) support the user. The first restriction means that, for example, validation of authorization information should not be performed at the client-side. The second restriction means that the focus of client-side form validation should be on user experience: clear information about the completeness and correctness and about the way a form should be filled in.

Robustness principle: The robustness principle [Postel 1981] says 'Be conservative in what you do, be liberal in what you accept from others'. A form is an interface with two faces: one to the application software and one to the user. Both faces have different requirements. For example, a software function might need a parameter of type integer, where a user might think in terms of 1, 1.00 or even 'one'; the form should – preferably – allow the user to enter 1, 1.00 or 'one', and the validation program should translate this input into an integer.
An interface satisfies the robustness principle if 1) no rigid demands on the user input exist, 2) automatic translation to internal representations occurs in those cases where the data entered is of a type that is unsuitable for the processing software, 3) the requirements on the data to enter are clear for the user, and 4) clear feedback is presented in cases of invalid as well as valid data (for example by showing a green check sign near the input field). Feedback should be presented after pressing the submit button.

Guide the user: A user should be guided implicitly as well as explicitly. Implicit guidance means using clear labels (specifying what should be entered), placeholders (presenting an example of what should be entered), fieldsets with legends (grouping related fields together), and, for example, radio buttons or drop-down lists when one item, or a limited number of items, should be chosen from an enumeration (limiting the number of possible values to enter) [Bargas-Avila et al. 2010].
Explicit guidance means that a user should be informed immediately after submitting the form about the completeness and correctness of the form.

Aim for reusability: Reusability means that software elements may serve for the construction of many different applications [Meyer 1997]; reusability is an obvious principle in case of form validation, because forms often ask the same data, for example, name, address, zip code, city, and phone number. As a result, often, the same validation has to be performed.

Aim for extensibility: Because variations in input data exist, the programming code should be easily extendible (new validation functions can easily be added) as well as adaptable (existing functions can easily be changed).

6.4.3 Programming rules for JavaScript

We supply our students with several programming rules for JavaScript. Examples are: 'Declare constants and variables before anything else.', 'Always use the keyword `var` to declare a variable and the keyword `const` to declare a constant.', 'Use capitals for constants.', 'Use object chaining if possible.', 'Put related functions and variables into a module, using the module pattern [Stefanov 2010], and compose a public interface with care.'

These rules help students to structure their code.

6.5 Procedural information

For the development of a form and the associated validation functions, we use a two layer architecture: 1) a GUI-layer, describing the static aspects of a form, specified in HTML5, and 2) a domain layer, describing the dynamic aspects of the form, specified in Javascript. For each of these layers, we provide guidance information. A data model, which we have to specify first, is used in both layers.

6.5.1 The data model

First, one should determine which data one wants from the user. One should specify both the data and user guidance:

Specify the data

First, we determine the type for each data item. We distinguish between string and integer (positive or negative whole numbers).

- In many cases, valid input values are limited to an enumeration. If this is the case, specify this enumeration.
- In case of a string with requirements on structure, specify this structure in the form a regular expression. Users are not supposed to enter numeric values according to the specific format they have in JavaScript. Instead, when users enter a decimal or another numeric value, the input is read as a string, and parsed against a regular expression. This allows us to adhere to the robustness principle.

  Not every format can be expressed in a regular expression. In such cases, the validation on the server-side will be more severe than the validation on the client-side. For validation on the client-side, regular expressions suffice.
- In case of integers, determine minimum and/or maximum values if applicable.
- In all other cases there are no requirements on correctness, which means that the format is 'free-form' text, for example, a text field to write an opinion about a certain product.

Specify user guidance

- Specify user guidance for each data item, to indicate how the form element should be filled in.
- Create an example value for each item.
- Create a label for each item.
- Specify, for each item, whether a value is required or optional.

6.5.2 The GUI layer

Create HTML5 elements

Based on the data model, determine a suitable HTML5 element for each input element, thereby satisfying the principle of implicit form validation. Examples are to use an HTML radio-button or drop-down list when the user must choose one item or a limited number of items out of an enumeration, or to use an HTML5 input element with type attribute `date` when the user should enter a date.

- Place a `span` element right to each `input` field. This element will be used to display feedback.
- Provide each `input` element with a `title` attribute containing the user guidance that you have specified in the data model.
- Use the example value for the `placeholder` attribute.
- Use the name for the item for the `label`.
- Give each `input` element a unique `id`.
- Give the `input` element an attribute `required` unless it is optional.
- Finally, in cases of requirements on structure or of integer values that should be within a range, specify the corresponding values for the `pattern` attribute (a regular expression) or the `min` and/or `max` attribute.
- Furthermore, group items together in fieldsets and take care of clear legends.

6.5.3 The domain layer

Although experienced developers may be able to design and implement code that will at once satisfy the design principles, less experienced developers, as students often are, can reach this level of design and code only by a step by step approach, in which refactoring is an important tool.

First, we give guidance on how to achieve a working application in the domain of form validation. Then we show the guidelines we give for every JavaScript application, to refactor such 'raw code' into an application that adheres to the design principles described before.

Create a form validation application

To create a form validation application, follow the following steps:

1. Specify and implement checking for completeness. To determine whether the form is complete, create a function that tests whether each `input` element with the attribute `required` has a value. Checking involves: read the data from the HTML `input` element, determine whether there is an attribute `required` and a value, and return `false` when a value is required but absent; otherwise, return `true`.

2. Specify and implement checks on correctness: depending on the specification of correctness of each `input` field, create a function that tests whether the value conforms to these specifications. This involves: read the data from the HTML `input` element, perform validation on correctness, write a feedback message to the corresponding `span` element, and return `true` or `false`. If there are no requirements with respect to correctness, just return `true`. If there is no value, this function should also return `true`: the check on correctness is only called if the check on completeness did tell that the field was either optional or contained a value.

3. Create an event handler for the `submit` button. This event handler first checks for completeness and then for correctness of all required fields. For now, the event handler prints a message to the console that tells whether the form is valid or not. In this event handler, one may also check on combinations of values (outside the scope of this article).

4. Check the code with a tool such as JSHint[2].

5. Create a test set and test the code.

Refactor the code

Unlike the rules that guide a student to create a form validation application, the guidelines for refactoring are general: these guidelines will be part of the procedural guidance in every JavaScript application.

The steps of these guidelines will sometimes have to be followed multiple times, depending on the result of the evaluation that follows this stage of refactoring. Refactoring is done by following these steps:

1. Remove duplicated code. If there are multiple instances of pieces of code, create helper functions to remove the duplication.

2. Think of different extensions that might be needed in the future, and check where changes would have to be applied in the code. If such an extension needs changes in different locations, try to restructure the code in such a way that the extension will become easier to apply.

[2] `http://www.jshint.com`

3. Check for cohesion within functions. Also, review the code with respect to sufficiency, completeness and primitiveness. If a function has more than one responsibility, split it in several functions.
4. Review the code with respect to separation of concerns. Separate the code that interacts with the DOM (this can be seen as the controller) from the code that contains the program logic.
5. Create modules for the controller and the program logic.
6. Review the modules with respect to encapsulation/information hiding and separation of interface and implementation. Only the functions that are needed outside the module should be exported; nothing else.
7. Review the code with respect to coupling. The model, for instance, should not need to call functions from the controller; only the other way around.
8. Check the code with a style checker tool such as JSHint.
9. Test the code with the same test set as was used for the original application.

Evaluate and document

The code should be evaluated with respect to the general design principles as well as to the specific principles for form validation. If the general and/or specific principles are not satisfied, the code must be refactored, along the previous guidelines.

Also, the code should be completed with documentation. Documentation can, of course, be written earlier, but at this moment, one must check whether the documentation is complete.

6.6 Applying the procedural information, first attempt

In this section we show how the procedural information, described in Section 6.5, indeed can lead to elegant code, using an example problem. For reasons of space, we show a small problem: a form consisting of two input elements. Both fields are part of a form for applying for a car insurance. The first field is a zip code field; the second field contains the number of years the applicant has driven without an accident.

6.6.1 The data model

Specify the data

For the zip code, the type of data is `string`. We specify that each zip code has to satisfy a structure constraint: a sequence consisting of four digits, an optional space, and two letters. In regular expression notation, the specification is:
$[1-9]\backslash d\{3\}\backslash s?[a-zA-Z]\{2\}$

For the `input` field representing the number of years without accident, the type of data is `integer`. Assuming a minimum value of `0` and a maximum value of `100` years, we specify this field as type `integer` with a minimum and maximum value.

Specify user guidance

As user guidance for the zip code, we specify: 'The correct format for the zip code is: four numbers between 0 and 9, optionally followed by a space, followed by two characters (either uppercase or lowercase). Two examples of the correct format for the zip code are 1234 AB and 1234ab'. The label for the item will be 'Zip code'.

As user guidance for the `input` field representing the number of years without accident, we specify: 'The number of years without an accident is a whole number between 0 and 100'. An example is `17`, and the name of the item will be 'Number of years without accident'.

Both items are required.

6.6.2 The GUI

Create HTML5 elements

The zip code field is specified as an input field of type `text`. The placeholder attribute shows a correct example.

The field representing the number of years without an accident is specified as an input field of type `number`. The placeholder attribute shows a correct example.

Both items receive the attribute `required`.

The HTML-code:

```
<label class="heading">Zip code</label>
<input id="zipcode"
       class="input"
       type="text"
       name="zipcode"
       pattern="^[1-9]\d{3}\s?[a-zA-Z]{2}$"
       placeholder="1234 AB"
       title="The correct format for the zip code is:
              four numbers between 0 and 9, optionally
              followed by a space, followed by two
              characters (either uppercase or lowercase)."
       required>
<span class="feedback"></span>

<label class="heading">
  Number of years without accident
</label>
<input id="yearswithout"
       class="input"
       type="number"
       name="yearsWithoutAccident"
       min="0"
       max="100"
       placeholder="17"
       title="The number of years without an accident is
              a whole number between 0 and 100."
       required>
<span class="feedback"></span>
```

Notice that each `input` element contains all information needed for validation: the value to validate and the specifications for completeness and of correctness, represented by the values of the attributes `pattern`, `min` and `max`. The responsibility of the script is to validate whether input of the user satisfies the restrictions specified in HTML; the responsibility of HTML is to specify which kind of input is asked for.

6.6.3 The JavaScript code

In this Section, we follow the guidelines presented in subsection 6.5.3, to show how students can achieve an elegant, flexible form validation module.

Create a form validation application

To illustrate how the refactoring guidelines work, we start with a 'naive' implementation, with a validating function for each of the form items.

First, we create a function that checks the completeness of an item, and writes a message in its feedback field in case of incompleteness. We use the JavaScript library jQuery. Here, we show how we declare all constants at the top of the code; in the next pieces of code, we omit the constants we already showed, for reasons of space. Likewise, we do not show tests and documentation.

```
1  const CLASS_FEEDBACK   = ".feedback",
2        INVALID          = "invalid",
3        MISSING          = "You probably forgot this entry",
4        REQUIRED         = "required",
5        VALID            = "valid",;
6
7  function isComplete(el) {
8    var fbField = el.next(CLASS_FEEDBACK),
9    complete = !el.prop(REQUIRED) || el.val();
10   if (!complete) {
11     fbField.removeClass(VALID).addClass(INVALID).html(MISSING);
12   }
13   return complete;
14 }
```

Note that the expression in line 9 exactly represents the definition of completeness: if a field is required, a value should be present.

To check the correctness of the zip code, we create a function that returns `true` if the value conforms to the regular expression specified by the pattern, or if there is no value. The function returns `false` if the value does not conform to the specification.

```
const OK      = "&#10003;",
      PATTERN = "pattern",
      TITLE   = "title",
      ZIPCODE = "#zipcode";
function zipCodeIsCorrect() {
  var isCorrect = true,
      element   = $(ZIPCODE),
      fbField   = element.next(CLASS_FEEDBACK),
      value     = element.val(),
      pattern   = element.attr(PATTERN),
      regex     = new RegExp(pattern),
      title     = element.attr(TITLE);
```

```
  if (value.length > 0) {
    isCorrect = regex.test(value);
    if (isCorrect) {
      fbField.removeClass(INVALID).addClass(VALID).html(OK);
    }
    else {
      fbField.removeClass(VALID).addClass(INVALID).html(title);
    }
  }
  return isCorrect;
}
```

Because of the specification of the HTML input elements, the code of other event handlers for validating correctness against a pattern is almost the same. Most of the code could be copied and only the `id` would have to be changed (which means we already see a reason to refactor later).

The implementation of the event handler validating the number of years without an accident could be as follows:

```
const MAX           = "max",
      MIN           = "min",
      YEARSWITHOUT = "#yearsWithoutAccident";
function yearsWithoutAccidentIsCorrect() {
  var isCorrect = true,
      element   = $(YEARSWITHOUT),
      fbField   = element.next(CLASS_FEEDBACK),
      value     = parseInt(element.val()),
      min       = element.attr(MIN) || Number.MIN_VALUE,
      max       = element.attr(MAX) || Number.MAX_VALUE,
  title = element.attr(TITLE);
  if (value.length > 0) {
    isCorrect = value >= min && value <= max;
    if (isCorrect) {
      fbField.removeClass(INVALID).addClass(VALID).html(OK);
    } else {
      fbField.removeClass(VALID).addClass(INVALID).html(title);
    }
  }
  return isCorrect;
}
```

At last, we create an event handler for the submit button, that prevents the browser from submitting the form, and checks whether the input is complete and correct. The application is now complete (apart from the fact that we do not include the code for the Ajax call to the server in the case when the input is complete and correct).

```
const CLICK = "click",
      SUBMIT = "#submit";
$(document).ready(function () {
  $(SUBMIT).on(CLICK, isSubmittable);
});

function isSubmittable(event) {
  var zipcodeComplete = isComplete($(ZIPCODE)),
      yearsWithoutAccidentComplete = isComplete($(YEARSWITHOUT)),
      complete = zipcodeComplete && yearsWithoutAccidentComplete,
      zipcodeCorrect = zipCodeIsCorrect(),
      yearsWithoutAccidentCorrect = yearsWithoutAccidentIsCorrect(),
      correct = zipcodeCorrect && yearsWithoutAccidentCorrect;
```

```
  event.preventDefault();
  return complete && correct;
}
```

6.6.4 Refactor the code

When reviewing the Java Script code from the previous subsection, it is clear that a number of general as well as specific principles are not sufficiently met. We will follow the guidelines to improve our code.

1. Remove duplicated code

The most obvious occurence of duplication is putting text in the feedback field and adding and removing a class. The guideline tells us to create a helper function. This function should receive information about the class (valid or invalid), and about the message that should be shown. To transfer this information, we create a constructor `Message` for message objects (line 1), and a function `createMessage` that receives a message object and an element to write to (line 5). We also write a function `handleMessage` that creates the right message, based on the value of `correct` (line 13).

```
1  function Message (valid, feedback) {
2    this.valid  = valid;
3    this.feedback = feedback;
4  }
5  function writeMessage(field, message) {
6    if (message.valid) {
7      field.removeClass(INVALID).addClass(VALID).html(message.feedback);
8    }
9    else {
10     field.removeClass(VALID).addClass(INVALID).html(message.feedback);
11   }
12 }
13 function createMessage(correct, field, errorString) {
14   var message = new Message(true, OK);
15   if (!correct) {
16     message.valid = false;
17     message.feedback = errorString;
18   }
19   writeMessage(field, message);
20 }
21 function isComplete(el) {
22   var fbField  = el.next(CLASS_FEEDBACK),
23       complete = !el.prop(REQUIRED) || el.val(),
24   if (!complete) {
25     createMessage(complete, fbField, MISSING);
26   }
27   return complete;
28 }
29 function zipCodeIsCorrect() {
30   var isCorrect = true,
31       element   = $(ZIPCODE),
32       fbField   = element.next(CLASS_FEEDBACK),
33       value     = element.val(),
34       pattern   = element.attr(PATTERN),
35       regex     = new RegExp(pattern),
36       title     = element.attr(TITLE);
```

```
37    if (value) {
38      isCorrect = regex.test(value);
39      createMessage(isCorrect, fbField, title);
40    }
41    return isCorrect;
42  }
43  function yearsWithoutAccidentIsCorrect() {
44    var isCorrect = true,
45        element   = $(YEARSWITHOUT),
46        fbField   = element.next(CLASS_FEEDBACK),
47        value     = parseInt(element.val()),
48        min       = element.attr(MIN) || Number.MIN_VALUE,
49        max       = element.attr(MAX) || Number.MAX_VALUE,
50        title     = element.attr(TITLE);
51    if (value) {
52      isCorrect = value >= min && value <= max;
53      createeMessage(isCorrect, fbField, title);
54    }
55    return isCorrect;
56  }
```

In lines 25 and 39, we now see a call to the function `createMessage` instead of two similar pieces of code that change the feedback field directly.

2. Review changes in case of extensions

The first possible extension that springs to mind, is to add another item to the form. In the application as it has been structured at this moment, a separate function should be written for each item. It would be nice if the application could be used for every form; not just for this particular one.

An input value is either a number (for which a range might be set) or a text which might have to obey certain rules. This means that there will be two general validating functions:

```
function isValidAgainstRegex(element) {
  var isCorrect  = true,
      fbField    = element.next(CLASS_FEEDBACK),
      value      = element.val(),
      pattern    = element.attr(PATTERN),
      regex      = new RegExp(pattern),
      title      = element.attr(TITLE);
  if (value) {
    isCorrect = regex.test(value);
    createMessage(isCorrect, fbField, title);
  }
  return isCorrect;
}

function isValidNumber(element) {
  var isCorrect = true,
      fbField   = element.next(CLASS_FEEDBACK),
      value     = parseInt(element.val()),
      min       = element.attr(MIN) || Number.MIN_VALUE,
      max       = element.attr(MAX) || Number.MAX_VALUE,
      title     = element.attr(TITLE);
  if (value) {
    isCorrect = isValid = value >= min && value <= max;
    createMessage(isCorrect, fbField, title);
  }
  return isCorrect;
```

```
}
```

We would like to rewrite `isSubmittable` in such a way that we can call one and the same function for each input. Note that, here, `isComplete` and `isCorrect` have side-effects to show feedback, and that the side-effect of `isCorrect` is only needed when the input is complete.

```
const INPUTS  = "input:not(#sumbit)";

function isSubmittable(event) {
  var complete = true,
      correct = true;
  event.preventDefault();
  $(INPUTS).each(function(index, element) {
    if (!isComplete($(element))) {
      complete = false;
    }
    else {
      if (!isCorrect($(element))) {
        correct = false;
      }
    }
  });
  return complete && correct;
}
```

Now, we have to implement a function `isCorrect`, which chooses, depending on the `type` of an item, which correctness checking function to use.

```
function isCorrect (element) {
  var res = true;
  switch (element.attr(TYPE)) {
    case RANGE  :
    case NUMBER : {
      res = isValidNumber(element);
      break;
    }
    default : {
      if (element.attr(PATTERN)) {
        res = isValidAgainstRegex(element);
      }
      break;
    }
  }
  return res;
}
```

3. Check for cohesion, sufficiency, completeness and primitiveness

With respect to cohesion: `isValidAgainstRegex` and `isValidNumber` write feedback as a side-effect. This has a negative effect on primitiveness: those functions could be given less responsibilities. An obvious place to write the feedback is in de function `isSubmittable`:

```
const INPUTS  = "input:not(#sumbit)";

function isSubmittable(event) {
  var complete = true,
```

```
      correct = true;
  event.preventDefault();
  $(INPUTS).each(function(index, element) {
    var el = $(element),
        feedbackEl = el.next(CLASS_FEEDBACK);
    if (!isComplete(el)) {
      complete = false;
      createMessage(complete, feedbackEl, MISSING);
    }
    else {
      if (!isCorrect($(element))) {
        correct = false;
      }
      createMessage(correct, feedbackEl, el.attr(TITLE));
    }
  });
  return complete && correct;
}
function isValidAgainstRegex(element) {
  var isCorrect  = true,
      value    = element.val(),
      pattern  = element.attr(PATTERN),
      regex    = new RegExp(pattern);
  if (value) {
    isCorrect = regex.test(value);
  }
  return isCorrect;
}

function isValidNumber(element) {
  var isCorrect = true,
      value     = parseInt(element.val()),
      min       = element.attr(MIN) || Number.MIN_VALUE,
      max       = element.attr(MAX) || Number.MAX_VALUE;
  if (value) {
    isCorrect = isValid = value >= min && value <= max;
  }
  return isCorrect;
}
function isComplete(el) {
  return !el.prop(REQUIRED) || el.val();
}
```

The validating functions now have no side-effects; the only function with a side-effect is `isSubmittable`, and the only purpose of that function is to have side-effects: to give feedback to the user, and, if the form is complete and correct, to send the form to the server. We might decide for a different name for this function, to show that it has side-effects, but for now, we leave the name as it is. As a bonus, we now do not need the constructor for `Message` objects anymore.

For now, the application seems to be sufficient and complete with respect to various form items. One exception is that we assumed that every item with type `range` involves integers, which means that the attribute `step` has the default value of 1. That attribute may also have a value of, for instance, 0.5 or 0.1. In that case, we would need another validating function, and `isCorrect` would have to be adapted. We will leave this for now, but remark that the set of guidelines help us in observing such an extension, which would make the validation module more generally usable.

4. Review with respect to separation of concerns

The separation between the HTML and the JavaScript is clean: in the HTML, the type of each input item is specified, and in some cases a minimum, maximum, or a pattern is specified as well. Also, the HTML specifies whether an item should be given a value. In the script, a validation function is chosen based on the specifications, and the required items are checked on the presence of a value.

Within the script, however, there is no separation of concerns. The first action could be to divide the code into two files. One file, `controller.js`, will contain everything that addresses the DOM (changing the DOM, binding event handlers), while the other file, `validator.js`, will contain the validating functionality.

The file `validator.js` is as follows:

```
const ...;

function isComplete(el) ...

function isCorrect (element)....

function isValidAgainstRegex(element) ...

function isValidNumber(element)  ...
```

If, in the future, more validating functions are required, the only function that will require a change is the function `isCorrect`. An alternative is to specify the kind of validation that is required in HTML.

The file `controller.js` is as follows:

```
const ...;

$(document).ready(function () {
  $(SUBMIT).on(CLICK, isSubmittable);
});

function isSubmittable(event)  ...

function createMessage(valid, field, message) ...

function writeMessage(field, message) ...

function Message(valid, feedback) ...
```

5. Create modules

Now, it is easy to create modules. The only functions that the controller needs of the validator are `isComplete` and `isCorrect`. The public API thus only consists of those functions.

```
var validator = (function (){
  // private
  const ...;

   var isComplete = function(el) {
    return complete = !el.prop(REQUIRED) || el.val();
  },
```

```
  isCorrect = function(element) {
   var correct = true;
   switch (element.attr(TYPE)) {
     case RANGE  :
     case NUMBER : {
       correct = isValidNumber(element);
       break;
     }
     default : {
       if (element.attr(PATTERN)) {
         correct = isValidAgainstRegex(element);
       }
       break;
     }
   }
   return correct;
 },

  isValidAgainstRegex = function(element) {
   var isValid  = true,
       value    = element.val(),
       pattern  = element.attr(PATTERN),
       regex    = new RegExp(pattern);
   if (value) {
     isValid = regex.test(value);
   }
   return isValid;
 },

  isValidNumber = function(element) {
   var isValid = true,
       value     = parseInt(element.val()),
       min       = element.attr(MIN) || Number.MIN_VALUE,
       max       = element.attr(MAX) || Number.MAX_VALUE;
   if (value) {
     isValid= value >= min && value <= max;
   }
   return isValid;
 }
 // public API
 return {
   isComplete: isComplete,
   isCorrect: isCorrect
 }
}());
```

The controller may also be changed into a module, and does not have to reveal a public API.

```
var controller = (function (){
  // private
  const ...;

  var isSubmittable = function(event) {
    event.preventDefault();
    $(INPUTS).each(function(index, element) {
      var el = $(element),
        feedbackEl = el.next(CLASS_FEEDBACK);
      if (!validator.isComplete(el)) {
        writeMessage(false, feedbackEl, MISSING);
      }
      else {
        if (validator.isCorrect(el)) {
```

```
          writeMessage(true, feedbackEl, OK);
          // send form to server
        }
        else {
          writeMessage(false, feedbackEl, el.attr(TITLE));
        }
      }
    });
  },
  writeMessage = function(valid, field, message) {
    if (valid) {
      field.removeClass(INVALID).addClass(VALID).html(message);
    }
    else {
      field.removeClass(VALID).addClass(INVALID).html(message);
    }
  }

  $(document).ready(function () {
    $(SUBMIT).on(CLICK, isSubmittable);
  });
}());
```

6. Review information hiding and separation of interface and implementation

By encapsulating the code into modules, and by providing the smallest possible public API, our code already conforms to these rules.

7. Review the code with respect to coupling

In this case, the controller calls two functions of the public API of the validator; the validator does not need anything of the controller.

Finally, we check the code with JSHINT and test the code.

6.6.5 Evaluate and document

Now, we are ready to review our code with respect to the design rules of subsection 6.4.1.

Abstraction: The validator exports two abstract functions, and the controller does not refer to individual form items. The level of abstraction is high.

Coupling and cohesion: The level of coupling is low: the controller uses two functions of the public API of the validator, and that is all coupling there is between the JavaScript files. There is coupling between the HTML and the script: the script relies on the `type`, `min`, `max`, `required`, and `pattern` attributes in the HTML. However, this is all within the HTML standard.

The level of cohesion of the validator is high: the validating functions only validate, and do not have side-effects.

Decomposition and modularization: The level of decomposition and modularization is high.

Encapsulation/information hiding: The level of encapsulation and information hiding is high, because the public API of the validator contains only two functions. The functions of the controller cannot be reached from outside because the code is encapsulated in a module.

Separation of interface and implementation: There is a clear separation, in the validator, between the interface (the public API) and the implementation.

Sufficiency, completeness and primitiveness: The level of primitiveness is high, because each function now carries one responsibility. The only exception is the function `isSubmittable`, in which feedback is written as a side-effect. We think that is allowable, because checking whether the form may be submitted to the server and giving feedback if that is not the case is so closely related. The code is sufficient for form application. With respect to completeness, the code should be extended for ranges with steps other than 1.

Separation of concerns: The level of separation of concerns is high.

Specific design principles: From the specific design principles for form validation, the first four (server-side validation is obliged, enough is enough, robustness, and guide the user) are met or could be easily met. The last one, aim for reusability, is met also: this code is applicable to all forms (with the proposed extension with respect to steps other than 1).

6.7 Discussion, conclusion and future work

Procedural information that help students solve complex problems is not available for many complex tasks in the domain of Computer Science. We developed such information for the task of programming in JavaScript. Programming in general is a complex task, and programming in JavaScript even more so, because there is no compiler to detect some types of mistakes, and the language allows one to write code that is hardly maintainable and difficult to analyse.

To validate the guidelines that we developed, we used them in an example task. We have described how our guidelines may help students to derive maintainable code that adheres to the software engineering principles.

We think that we should create step by step guidelines for the many complex tasks that we prepare our students for. This set of guidelines is one example. In our opinion, these guidelines are not simple-to-follow recipes that guarantee success. Instead, these guidelines help students to divide problems into smaller ones, to stimulate an attitude of first thinking, then doing.

The work that we have done not only shows that our guidelines might be helpful; we have also created an example case to explain our students how to use these guidelines.

What needs to be done, however, is to monitor whether these guidelines really help students. It is clear that the guidelines *may guide* students toward a good enough solution, but we will have to check whether they really *do so* in practice. We are currently working on a setup that will allow us to validate the working of

these guidelines. In a situation of distance learning, we might use the Think aloud method in a session in our electronic learning environment, which allows us to record what the students does and says.

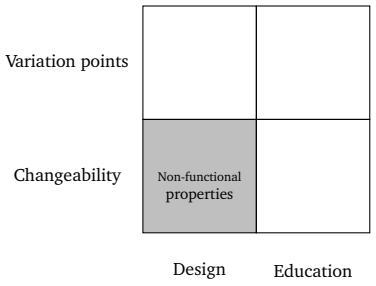
In the far future, we would like to work on tools to check automatically if a solution is good enough. Our guidelines could be used to provide meaningful feedback when a solution is not yet acceptable.

Part II

Enhancing changeability

7

Software architecture and non-functional properties[1]



This article was written during the time that the field of software architecture began to emerge.

In this article, we demonstrate that the choice for a software architectural style influences non-functional properties of the system such as extensibility.

We also show that the choice for a modeling technique (actions and events in the form of a finite state machine or interacting concurrent processes in the form of PAISLey) to model the desired behavior of the system is intertwined with the choice for a software architectural style: the choice for a modeling technique influences the structure of the architecture.

This observation has implications for projects to generate code from specification, by translating requirements automatically into a behavioral model and translate that into code [Toetenel et al. 1996]. One chooses, implicitly, a modeling technique when implementing the translation of the requirement specification into a behavioral model. When doing so, one places restrictions to the style of the resulting architecture, without making explicit choices.

[1]This article originally appeared under the title of 'Evaluation of Software Architectures for a Control System: A case study' in Coordination Languages and Models, the Proceedings of the Second International Conference COORDINATION, pages 157–171, 1997.

Relevance to this thesis

The relevance to this thesis is that architectural styles differ with respect to changeability, and that the choice for a modeling technique to model the desired behavior of the system is intertwined with the choice for an architectural style. The choice for such a modeling technique therefore, in the end, influences non-functional properties such as changeability.

Relevance today

Nowadays, it is common knowledge that influence of the software architectural style on non-functional properties exists: several researches, including ours, have demonstrated this influence.

The emphasis within the field of software architecture was, at that time, on the structure of a system. Later on, the behavior of a system became one of the aspects that are described by a software architectural document, but in these early days, the sole emphasis was on structure.

Even though a description of the behavior of a system has become part of its architecture, the observation that the modeling technique for the behavior of a system influences the structural style, is not often made today.

With respect to the non-functional properties that we check, we observe that today, security would certainly be one of them.

Abstract

In this paper, we give our view on the software architecture phase in the development process. During this phase, we distinguish modeling and structuring activities. A system is modeled according to a certain approach, and this model is used to instantiate a certain architectural style. In general, the activities are intertwined.

The choice for a specific software architecture has implications on the non-functional properties of the system. We illustrate our view with a case study of a software controller for a (toy) railroad system that we have available in our software lab. Several models of this system, expressed in formal specification languages, were made in the past, so we are able to produce a software architecture for the system while carrying out both activities separately.

The resulting software architectures are evaluated with respect to timing aspects, scalability, fault-tolerance, and extendibility. Extendibility of a software system is especially important for domains where changes should be applicable at run-time. Design for change should start at the software architectural level.

7.1 Introduction

In this article, we illustrate our view of the software architecture phase in the development process and the implications of the choice for an architectural style within that phase, with a case study of a railroad system. The essence of our view is that

we distinguish modeling and structuring activities. The choice for a specific software architecture has an impact on the non-functional requirements of the system. Therefore, when evaluating different architectures for a certain system, one should take into account the non-functional properties that are relevant. Roughly said, one addresses the functional requirements during the modeling activities, and the non-functional requirements during the structuring activities.

A software architecture-driven development process consists of a Requirements Analysis phase, a Software Architecture phase, a Construction phase, and a Maintenance and Change phase. During the Software Architecture phase, one models the system, chooses a software architectural style, instantiates this style, and refines the instantiation either by adding detail or by decomposing components or connections (again going through modeling, choosing a style, instantiation and refinement). This process should result in an architecture that is defined with so much detail that either reusable components and connections can be fitted, or components and connections can be designed and implemented. Note that this view does support a sequential as well as an iterative or an incremental development process, and that in general, the modeling and structuring activities are intertwined.

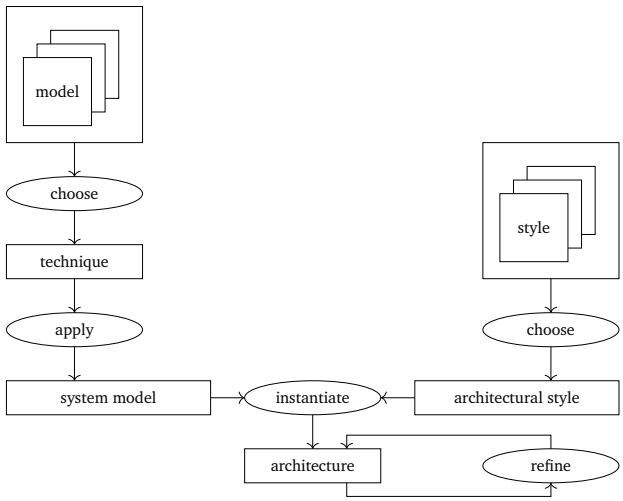


Figure 7.1: Software architecture in the development process

The Software Architecture phase as we view it is depicted in Figure 7.1. In this figure, ovals denote activities, while boxes represent products. Input for all activities are the requirements (not shown). At the left, one chooses a modeling technique and applies this technique to the problem at hand, while at the right, one chooses a software architectural style. The style is instantiated with the model of the system, and the result is an architecture that is refined.

An architectural style is a pattern in the organization of software [Shaw and Garlan 1996], or, somewhat more precisely, 'A set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done' [Shaw and Clements 1997]. Architectural styles are categorized in taxonomies in order to provide guidelines mapping classes of problems onto classes of solutions [Shaw and Clements 1997].

Boasson argues that at the highest level, two fundamentally distinct approaches towards software architectures can be discerned: the data-centered and the function-oriented approach [Boasson 1995]. To relate this statement with our view of the Software Architecture phase in the development process, one could say that a data-centered or a function-oriented model each leads to different sets of software architectural styles.

We describe two software architectures on a high level of abstraction, based on two different approaches for system modeling for a (toy) railroad system that we have available in our software lab. We have a data-centered model of the system, described in the formal specification language AE-VDM [Biegstraaten et al. 1994]. We also have a function-oriented model of the railroad system, described in PAISLey [van Katwijk and Toetenel 1995]. The railroad network described in these models differs from our toy railroad system at some points, so we had to adapt the models.

For each architecture, we derive the implications they have on those quality properties that are important for a railroad controller:

Timing : The requirements of a real-time system usually contain temporal constraints. In the case of the railway network, there are strict temporal constraints because of safety reasons, and less strict temporal constraints with respect to the schedules. Performance with respect to these constraints can only be measured when all design decisions have been made. It is highly desirable to decrease this gap between temporal constraints and performance at the architectural level. We discuss time aspects of each proposed architecture.

Scalability : Both the toy railroad system and the railway network for which the models were originally developed, are scale models of real-life situations. Scalability is a requirement for a software architecture.

Fault-Tolerance : One of the problems of controlling a physical system is that such a system often does not behave exactly according to whichever model we use to represent it. Reliability addresses the behavior of a system in an environment behaving according to the model; robustness addresses behavior of the system in 'abnormal' circumstances. Behavior in abnormal circumstances is often indicated with the term 'incident handling'. In each architecture, we indicate which changes are needed to incorporate incident handling in order to achieve a certain degree of robustness. Incidents are not only formed by unexpected events in the railway network, but also by failing communication and hardware.

Extendibility : Requirements are not as static and final as they are usually treated. They change, either as a result of an inaccurate modeling of the environment

or of a changing world. The answer to changing requirements is a changing system. In a system like the railway network presented here, it is necessary to apply changes at run-time.

Two possible mechanisms for changes at run-time of software found in literature are:

- A change at the architecture level, consisting of adding or destroying components and connections. A model for 'dynamic change management' along these lines is presented by Kramer and Magee [Kramer and Magee 1990].
- A change at source code level. Frieder and Segal described a scheme for procedure replacement [Segal and Frieder 1988].

In our opinion, design for change should start at the architectural level. When evaluating an architecture, one should bear in mind that the first mechanism should be applicable in the changes one can think of.

The two architectures are proposed and discussed in Section 7.2 and 7.3 The final section contains conclusions and suggestions for future research.

7.2 Data-centered approach: a global state architecture

The first type of architecture that we analyze is based on a data-centered model of the railroad controller for our toy railroad system [Biegstraaten et al. 1994]. The solution given below is meant as an example of the global state architecture; we do not pretend to propose an optimal solution.

7.2.1 Event-action model

According to e.g. Parnas [Parnas, van Schouwen et al. 1990], the behavior of reactive systems can successfully be modeled in terms of events and actions. Events can be defined in terms of changes in the global state of the system, including time. Actions consist of computations resulting in changes in the global state. Similarly, the functionality of the railway system can be described in rules, specifying an action for each discerned event.

In the first place, the speed behavior of each individual train is modeled by a finite state machine, shown in Figure 7.2. In state `HALT`, a train is stopped (temporarily). State `ACC` is the state of a (gently) accelerating train; state `DEC` is the state of a gently decelerating train. A train in state `CONST` drives with a certain constant speed. A train in state `EM` (emergency) stops as soon as possible.

The transitions in the finite state machine are described by action-event rules, stating the events that trigger a transition (Table 7.1). These events involve information about the desired speed for each train, to be generated on-the-fly from the schedule of the train. The 'before' column shows the state before the transition; the

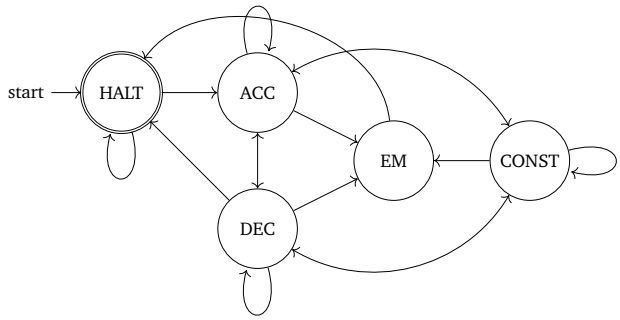


Figure 7.2: State diagram for a train

BEFORE	EVENT	AFTER
HALT, CONST, ACC or DEC	desired speed > actual speed	ACC
HALT or DEC	desired speed = actual speed = 0	HALT
ACC, CONST or DEC	desired speed = actual speed	CONST
ACC, CONST or DEC	desired speed < actual speed	DEC
ACC, CONST or DEC	state of train is ERROR	EM
EM	actual speed = 0	HALT

Table 7.1: Speed of a train

'after' column the state after the transition; the 'event' column describes the event that triggers the transition.

Another finite state machine is used to model the overall behavior of a train (Figure 7.3).

Five states are discerned: in the `STAT` state, a train is situated at a station; in state `START`, the route to the next station is (being) determined; in state `GO`, the train is driving; in state `WAIT`, the train is stopped somewhere along the route; state `ERR` is used for cases of failures. We have assumed that in the initial state, a train is always positioned at a station. The transitions are shown in Table 7.2.

Data such as the desired speed is set as a side-effect of state transitions. Table 7.3 shows a simple way of setting the desired speed. Other side-effects consist of determining the route to be taken, and the setting of switches in the railway network.

Figure 7.4 shows an architecture based on this model. A central data store component contains the relevant data and sends events, representing changes in the state or time, to components acting upon these events. These components are able to read and write the data. The proposed architecture can be seen as an instantiation of the blackboard style [Shaw and Garlan 1996].

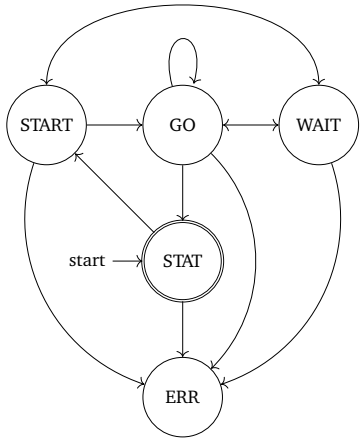


Figure 7.3: State diagram for a train

BEFORE	EVENT	AFTER
STAT	departure time reached	START
START or GO	next part of route free	GO
START or GO	next part of route blocked	WAIT
GO	destination reached	STAT
WAIT	next part of route free	GO
WAIT	deadlock occurred	START
all states	error occurred	ERR

Table 7.2: Behavior of a train

STATE TRANSITION	DESIRED SPEED
START → GO	maximum speed
GO → WAIT	0
GO → STAT	0

Table 7.3: Setting the desired speed

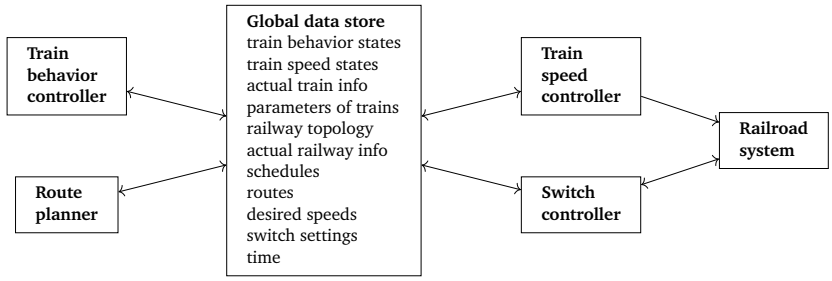


Figure 7.4: Global state architecture

The `Global Data Store` is used to store the global state. Essential in this architecture is the fact that all data are stored globally. As a result, all data needed by components are found in the global data store.

Information kept in the global data store consists of the state of the behavior and speed of each train, of the actual information (about speed and position) of each train, the parameters of the trains, the schedules, the derived routes and desired speed, the topology of the railway, the switch settings, and the time.

Certain transitions in the global state represent events.

The `Train Behavior Controller` carries the responsibility of maintaining the finite state machine representation of the behavior of the trains, according to the rules described in Table 7.2. The information it needs consists of the train behavior states, the schedules, time, the positions, speed and directions of other trains, and the train parameters. The component modifies the train behavior states, the desired speed, and the switch settings.

The `Route Planner` is responsible for determining the route to be taken to the next station mentioned in the schedule of the train. One may implement a deadlock avoiding route planner, or one that does less planning ahead. In the last case, a second task of the route planner is deadlock detection (and, as a consequence, determine new routes).

Information needed by the route planner consists of the schedules, the railway topology, and of the positions, speed and direction of the trains.

The `Train Speed Controller` is responsible for maintaining the desired speed in a comfortable way, according to the rules in Table 7.1. Another task of this component is to update information about the actual position, speed and direction of the trains.

Changes in the desired speed of a train and the transition to behavior state `ERR` form events of interest for the train speed controller.

The `Switch Controller` has the task of updating information about the state of the switches and setting them.

Multiplicity of components: The architecture as it is proposed here does not state anything about the multiplicity of the components. Obviously, there is only one

data store. On the other hand, each train might have its own behavior and speed controller, and route planner. Multiplicity of the switch controller is a possibility as well. Multiplicity of the behavior controller, speed controller, route planner and switch controller is an open design decision in this architecture.

7.2.2 Implications on non-functional properties

Timing

To analyze the timing behavior of a system implemented along these lines, the components performing the functionality should be implemented as cyclic, asynchronously communicating processes. These processes poll the global data store to obtain information about the relevant data. As a consequence, restrictions to the cycle time can be derived from the temporal constraints and the speed and duration of the connections and computations.

Scalability

In the case that routes for the trains are generated decentrally, on the fly, the possibility of deadlocks is present. With an increasing number of trains driving on a railway network, deadlocks will occur more frequently. The introduction of deadlock avoidance may become necessary, though this will have implications on the timing aspect. However, whether deadlock avoidance is chosen or not, the architecture as we have presented it here suits both solutions. Because the global state contains the data of all trains, a deadlock avoiding algorithm can be introduced very easily.

For scalability reasons, it should be possible to parallelize the computation. As we have seen, the train behavior and train speed controller can be parallelized (one for each train). The train speed controller can be split into a component maintaining the speed, and a component polling the train for actual speed, position and direction information.

The route planner might be parallelized as well, but in that case, deadlock avoidance is better performed by a separate component. In both cases, computation time increases with an increase of the complexity of the railway network.

Another possible bottleneck is formed by the access of the global data store. Introduction of (parallel) agents detecting changes in parts of the state, with the possibility to read and write data, might be needed with an increase of the railroad system.

Extendibility

Changes in the topology of the network are introduced as changes in the data of the global state. A component, responsible for deriving new schedules, might be introduced. In this case, the timing issue (components should not make use of the new data too soon or too late) is rather trivial: a physical change in the railway network topology will always take place with trains at a safe distance, so the new

situation will be read by the controller components by the time that a train has reached a new situation.

Changes in the parameters of trains are to be introduced in the same way, by changing the data in the global data store. When the changes are applied when the train is in state STAT at a station, the new parameters will be used in time.

The same applies for changes in the schedules of trains: the schedule is read by the route planner component when the train is going to leave the station, so the new data will be used in time.

In general, the conclusion is that the proposed architecture is an easily extendible architecture. Data can be changed fairly easy, and an extension of the functionality can be done by adding or changing components, and adding or changing data in the global data store. No big changes in the architecture are needed, because components never communicate directly.

Prerequisites are that changes in the global data store can be applied from outside, and that components and new data and datatypes can be added at run-time.

Fault-tolerance

A failing train should result in an emergency stop. In our model, this will be effectuated when the event `ERR` occurs in the global data state (Table 7.2). Error-detection might be an extra task of the train speed controller (actual speed differs too much from the expected speed), or may be performed by a new component.

Incident handling requires an overall view of the system. In the global state architecture, each component conceptually has such an overview. As a result, one can add components with intelligent incident handling capacities fairly easily.

A failing communication network is another source of problems. We can discern different type of data in the data store: information that is updated frequently, such as the actual position, speed and direction of the trains, and information representing a state, where each change is a major difference with the old data.

The loss of messages containing the first type of data is not really a problem: as long as the time restrictions are not too tight, a decision based on information that is slightly older than it should be will do no harm.

Messages containing information about an event or a state transition should not get lost. A solution to prevent such loss might be to handle this kind of information in the same way as continually updated information: instead of waiting for an event, components poll the data in the global data store. Each time when they poll, they update the state information (or data changed as a side-effect) in the global data store.

To make the system fault-tolerant, the global data store should be duplicated and/or distributed on different hardware. In the case that all information is stored in the global data store, setting an extra processor with one or more components at work when another fails is easy, because local data do not exist.

7.3 Function-centered approach: a data flow architecture

The second software architecture is loosely based on a PAISLey model for the railroad controller, described in [van Katwijk and Toetenel 1995]. This PAISLey model is based on two computation models: asynchronously interacting concurrent processes, and functional programming. A specification written in PAISLey requires a process structure and a definition of the structure of interprocess communication, and can therefore, be mapped almost directly onto a software architecture.

The PAISLey specification of the railroad controller consists of cyclic, asynchronously communicating processes.

7.3.1 Software architecture

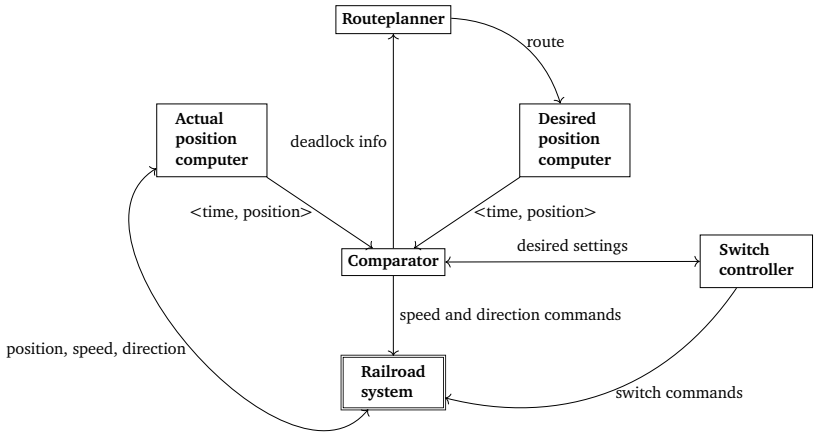


Figure 7.5: Dataflow architecture

The dataflow architecture depicted in Figure 7.5 is an instantiation of the control-loop architectural style [Shaw 1995]. The position of a train is the process variable to control. The actual position is compared to the desired position, and differences between these positions trigger speed or direction commands. The desired position is computed by consulting the route (which contains time information) and the parameters for the train. The actual position is obtained by polling the trains.

The `Route planner` computes the route for the trains. This component contains the schedules as local data. In the case of a deadlock, the route planner gets a message from the `Comparator`, whereupon it computes new routes.

The `Comparator` compares the actual and desired position. To simplify comparison, positions are attributed with an indication of time. A difference between the actual and the desired position at a same time indicates the need for control, to be executed in the form of speed and direction commands for the train, and sometimes commands to set a switch.

Information needed by the `Comparator` consists of the position and speed of a train and the state of the switches in the railway network.

The `Actual position computer` polls the trains and derives `<time, position, speed>` for the trains.

The `Desired position computer` derives the desired position for each train from the route, sent by the route planner.

The `Switch controller` keeps track of the state of the switches in the railroad, and sets them according to the messages of the `Comparator`.

Multiplicity of components: Both position computers and the comparator in this architecture can have multiple instances, i.e. one for each train. For the route planner, this is less obvious. Introducing deadlock avoidance in a system with one route planner for each train will be difficult, and will require severe communication between the different components. In a system where the routes for all trains are computed by one component, the information needed to avoid deadlocks is available at the right place.

7.3.2 Implications on non-functional properties

Timing

The comparator and the switch controller are triggered by both position controllers, which are the 'drivers' of the system. From the cycle time of the processes, the speed of the connections, and the time needed for different computations, one can analyze whether the system will respect the temporal constraints.

Again, there we should choose between deadlock avoidance or detection. Deadlock avoidance should be performed by the route planner. In that case, there should be one route planner for the whole system.

The most logical place for deadlock detection is the comparator, because it receives information about the position of each train. However, the computation time needed for deadlock detection might conflict with the temporal constraints for the comparator. In that case, an extra component should be introduced.

Scalability

As we have seen, parallelization can be introduced for both position computers and for the comparator. Inherent to this solution is that the computation time of the comparator increases with an increase of the number of trains: to determine whether a train is able to go on or should stop, the comparator needs information about the speed and position of all trains. An extra component, filtering the relevant information for the comparator, might be needed when upscaling the system.

The same applies for the route planner: its computation time increases with an increase of the complexity of the railway system.

Extendibility

A change in the railway topology network should be applied to data in all components. Changes in the parameters of a train should be applied in the comparator for the train. New schedules are to be added in the route planner.

In general, because data and computation are intertwined in this architecture as opposed to the previous one, one should, for each change in data, determine which of the components make use of the information. In the case of an extension of the functionality, one should determine which information is needed for an added component, and from which components this information can be derived.

Therefore, changes are inherently harder to apply than in the previous architecture.

Fault-tolerance

A failing train can be detected by the comparator (because of an increasing difference between the actual and desired position). Because the system is based on the comparison between the desired and the actual situation, no extra measures are needed to take failing trains into account. The comparator is the component that has an overall view of the system, because it receives speed and position information of all trains. However, this component should not be charged with incident handling, because it should perform under strict temporal constraints.

Adding incident handling is another case of adding functionality, and as has been said above, this is less straight-forward in the dataflow architecture than in the global state architecture, because in this case, components communicate directly, and data and functionality are not separated.

In some cases, the connections between the components are of the dataflow type (continuously updated information): this is the case for the connection between the railroad system and the actual position computer, and for the connection between both position computers and the comparator. The other connections are used for commands, or for information that is delivered once (a new route, a deadlock situation). When these connections fail to deliver a message, the result may be a disaster. A solution is to deliver these kind of messages multiple times.

Failing processors in this system are harder to replace than in the previous architecture. Every component has local data. The only way to be able to replace a processor is to keep track of these local data on redundant hardware.

7.4 Conclusions and future work

7.4.1 Conclusions

Software architecture in the sevelopment proces

In this article, we illustrated our view on the software architecture phase in the development process, sketched in Figure 7.1, with a case study of a railroad controller in software. Concerning the development process, we make the following remarks:

- In this case study, the software architecture phase was carried out sequentially: modeling the system took place first, and then a style was chosen and instantiated. The reason for this order was that several models of the system were already available.

  In general, system modeling and the choice and instantiation of an architectural style are carried out at the same time.

- The case study clearly shows the existence of relations between the activities within the software architecture phase. The choice of a model influences the choice of an architectural style, and vice-versa. The adequacy of different approaches toward system modeling for different architectural styles should be added to taxonomies of styles.

- Architectural styles differ more in the degree with which they satisfy the non-functional requirements than the functional requirements.

Implications of architectural styles on non-functional properties

Here, we summarize the effects of the proposed architectures on the quality properties that we found important.

Timing : Whether temporal constraints can be met or not can only be determined in a fully implemented system. In both architectures, we were able to reason under which conditions timing analysis would be possible, and we could reason about possible bottlenecks. At the level of abstraction of both proposed architectures, differences between the solutions with respect to timing issues cannot be found.

Scalability : A big difference between both proposed architectures is that in the global data store architecture, information is always available to every component. As a result, it is easy to divide the functionality of one component between several others. In the dataflow architecture, when dividing one component into several components (either with different functionalities or to have different components control different parts of the controlled system), one should always bear in mind how the newly created components get their information.

Fault-tolerance : Introducing incident handling requires on the one hand the possibility for a component to run in a separate thread, and on the other hand the possibility to gather information about the global state of the system. In the global state architecture, this requires a decision for multiple threads. In the dataflow architecture, multiple threads are part of the style. On the other hand, extending the functionality in this architecture is more difficult, because components communicate directly (so one has to determine where the necessary information should be obtained).

Failing hardware is handled more easily in the global state architecture, because the state is always available. The global state itself, however, should be duplicated.

Extendibility : Changes of data (topology of the railroad network, parameters of the trains, schedules) are very easy to apply in the global data store architecture. In the dataflow architecture, one should always determine which of the components store such information locally.

In general, changes to the functionality of the system are much easier to apply in the global data store architecture, because there is no need to analyze where the information, needed for each component, is to be obtained. A requirement for the possibility of changes at run-time is that it should be possible to add components, data and data-types at run-time.

7.4.2 Future directions

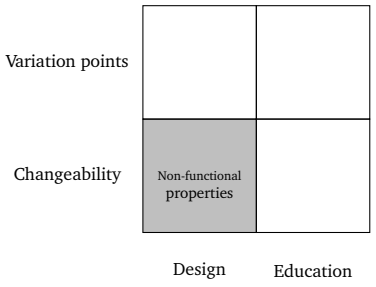
System evolution at run-time in real-time systems is considered one of the future challenges in this area [Stankovic 1996]. A promising approach would be to explore the possibilities of the global state architecture with this respect. Changes in functionality within this architecture can be applied by adding or substituting components. In addition, facilities to change the global state, and the generation and distribution of events, should be developed.

Even without facilities to change the global state, it is comparatively easy to handle failing processors when using the global state architecture, assuming the global state component is fail-proof: components performing computation may be substituted by other components without a loss of data.

The description of an architecture evokes a static view of components and connections. Architectures with possibilities for system evolution at run-time are dynamic. Apparently, techniques to describe and analyze the dynamics of architectures are lacking. Representation of the dynamics of architectural styles and instantiations form an interesting subject for future research.

8

A framework around the radio broadcast paradigm[1]



This article describes a framework for the systematic development of distributed real-time systems. In this approach, one software architectural style is used: the Radio Broadcast Paradigm (see Chapter 4).

The framework offers a language in which the resulting system can be described in a formal way. Such a description offers the possibility to prove that the system has certain desired properties. Because the platform on which the distributed system will run may vary, analysis is also used to derive constraints for those properties of the platform that influence the desired properties of the system as a whole.

This article shows that the choice of an architectural style and the choice for an implementation of that style, enables one to reason about desired properties of the resulting system.

Relevance to this thesis

The architectural style that is chosen for this framework, together with the formal language that can be used to describe the resulting system, allows one to reason about desired properties of the system. The architectural style is the Radio Broadcast

[1]This article originally appeared under the title of 'Software Development and Verification of Dynamic Real-Time Distributed Systems Based on the Radio Broadcast Paradigm' in Parallel and Distributed Computing Practices, pages 105–126, 2001.

Paradigm. The possibility to apply changes at run-time in this style is discussed in Chapter 4.

Deviations from the original article

In Section 8.2.2, we added some text that had to be removed in the original artciel because of the limiation on the number of words.

Abstract

The combination of an increased power of computer systems and a marriage between computing and communication causes an enormous increase in the complexity of applications in almost all domains. This also applies to the real-time and the embedded domains: construction of distributed applications is a major research area.

In our research, we develop a framework for the systematic development of distributed real-time control applications. We emphasize the use of pragmatic, sound approaches in the design steps of the development process, preferably based on some common architectural style.

For analyzing and validating critical elements of design and implementation, we emphasize the use of formalisms, however. Complexity of applications is such that for real verification and validation, proof or model-checking techniques are required. We use formalized abstraction as a technique for obtaining the appropriate templates from design and implementation. These abstractions are then dealt with in a model-checking and analysis tool set.

In this article, we show some elements of our approach, in particular we describe the architectural style that we are using, the Radio Broadcast Paradigm, and we demonstrate the viability of our approach by presenting a case study.

8.1 Introduction

The continuously increasing power of cheap computers and the integration between computer systems and communication mechanisms both have an enormous impact on applications. Applications become more and more distributed, obviously in the areas of business computing, but also in technical and control domains.

From our perspective, measurement and control applications are interesting since they depend, perhaps stronger than other kinds of applications, on the notion of time and on the reliability of the underlying networked systems. In distributed environments, these notions are hard to control, which influences design and implementation of distributed control applications considerably.

Part of the additional care to be given to the design of distributed control applications is based on the uncertainty in the stability of the environment: message passing in a network may not take a constant amount of time, networks may break down or be altered dynamically. Development of an application therefore should take into account an accurate characterization of the execution attributes of the underlying

infrastructure. Characterization of the underlying infrastructure and indicating requirements of the infrastructure are therefore integrated elements in the development cycle of this kind of applications.

The objective of our research is to establish a framework for the systematic development of distributed real-time control applications.

In our approach, practical development starts with the selection of an architectural style and an architecture that will fit the needs of the application. The selected architecture is supported by technology implementing the functionality of the architectural components. Such designs are gradually transformed into program code through the coding phase of system development.

The design process is supported by design analysis and simulation feedback. Analysis provides data that enables a more precise verification effort.

A classical example in this respect is the validation of timing properties of a system. Standard schedulability analysis provides a technique for analyzing whether or not real-time requirements are met, under the assumption that accurate execution characteristics are available. Here, testing and simulation can provide accurate timing data that, together with an abstract representation of the process structure of the implementation, form the basis for formal analysis and simulation.

In our approach, we use abstraction as a technique to derive the appropriate templates from design and implementation as an input to formal verification (see Figure 8.1).

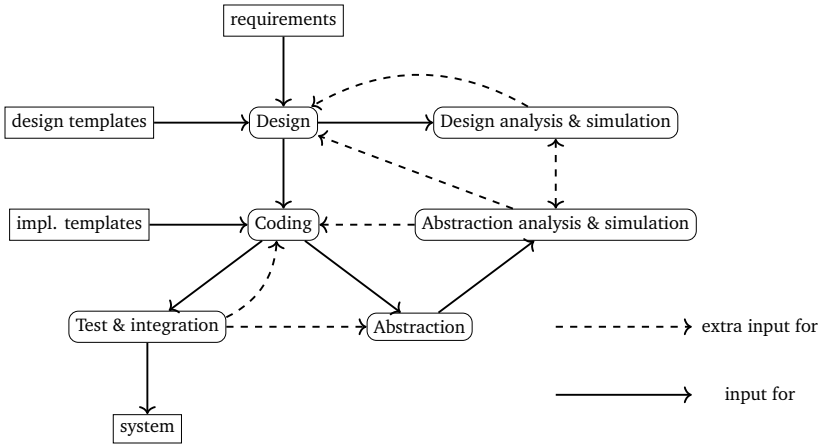


Figure 8.1: A Software Development Process

In order for our approach to be successful, we need to be able to build models of the technology supporting the architectural models, to be used in the validation of the architectural instances.

Analysis and verification can be addressed in two ways. The classical way is that testing and analysis of implementations provide data on execution characteristics, after which the application is analyzed using these data. For many distributed applications this approach is not appropriate, simply because the application runs on different underlying systems with different characteristics. An approach in which constraints on the architectural support mechanisms are derived, is desirable. In such an approach, analysis of the application leads to a characterization of essential constraints in the underlying system that, when met, guarantee the application to operate correctly.

One of the key issues in our research is therefore to build models for both applications and underlying frameworks with which a qualitative and quantitative analysis of execution parameters and attributes in distributed real-time control systems is possible. To limit the scope of our research we use Java as our language of choice, and we use a radio broadcast model for interprocess communication.

For the latter, we developed a standardized implementation and a number of specification and analysis tools. Part of the research then focuses on the characterization of the radio broadcast model implementation with respect to applications running on it.

In this article, we describe some elements of our development approach and discuss a (simplified) example of the analysis we pursue. The architecture we have chosen is the Radio Broadcast Paradigm (RBP), while the analysis technique used is parametric model checking. In Section 8.2 we describe the RBP and briefly discuss its implementation in Java. In Section 8.3 we describe a case study, a distributed control application. In Section 8.4 we describe our analysis approach and the results of the application to a simplified model of the distributed control application. Finally, in Section 8.5, we summarize our results and describe our current work.

8.2 The radio broadcast paradigm and its implementations

8.2.1 Subscription-based communication

A subscription-based communication model abstracts from the physical communications structure by introducing communication 'channels'. A process interested in specific information 'subscribes' to a channel. The real-world analogy to this action is tuning a radio to a certain frequency. After having subscribed to the channel, the process will receive all messages sent to the channel. Similarly, if a process wants to make information available to other processes, it can 'publish' the information on a channel, either an already existing one or a newly created one. In principle, the number of available channels is unlimited.

The properties described above make the subscription-based model appropriate for (data-dominated) embedded systems, which deal with a continuous information stream from the environment [Boasson 1996].

In most cases, the subscribers will be able to deal locally with an absence of incoming messages (temporary or permanent). The fact that there are no hard-coded dependencies between processes opens the possibility for change at run-time. Pro-

cesses can be added, removed, and/or replaced without disturbing the information flow. The subscription paradigm thus provides a mechanism for very late binding.

8.2.2 Existing models and implementations

The subscription paradigm is a form of multi-cast communication, for which there are many implementations [Coulouris et al. 1994]. In this section, we briefly mention a number of existing implementations.

Splice

SPLICE (Subscription Paradigm for the Logical Interconnection of Concurrent Engines) is an implementation of the paradigm developed by Boasson [Boasson 1996] of Hollandse Signaal Apparaten. The main application area is embedded systems for command and control systems.

SPLICE extends the basic subscription-based communication model as described above, by providing a conceptual global data space. This data space is partitioned into data sorts (channels). Data sorts are typed; the structure of data elements (their fields and field types) must be specified before the data sort is used. In addition, it is possible to mark fields of the data sort as key fields. The global data space is a virtual one; it does not physically exist.

Each of the processes in the space is extended with an *agent*, responsible for the communication with other processes. When a process subscribes to a data sort, the agent creates a local database, specific to that process. Whenever data is received from the network, it is stored in the local database. If a data item with the same values for the key fields as the newly arrived data item, is already present in the database, it will be overwritten. The process can use the local database as a 'mirror' of the global data space. The process can subsequently access the local database at any time for data matching a query.

SPLICE provides a rich query mechanism, so applications can retrieve exactly the data they are interested in. In addition to queries, processes can put a filter on a local database. Whereas queries operate on the information already present in the local database, a filter is used to decide whether a newly received data item is stored in the database. Filters enable a process to specify a more fine-grained selection of the information it wishes to process, and reduces the size of the local database.

In most cases, in particular when dealing with an information stream from the environment, starting with an empty local database is no problem. It will gradually be filled with information over time. In other cases, the local database must be brought up-to-date with information that was communicated before the process subscribed to a channel. SPLICE provides facilities to handle both cases.

SPLICE does not provide an operation to remove a data item from the global data space. Processes are allowed to remove items from their own local databases if they do not need the specific data item anymore, but a global remove-operation is not available.

JavaSpaces

JavaSpaces is a package of Java classes and interfaces, developed by Sun Microsystems very recently [Microsystems 1998]. The package is meant to support the design of distributed applications using Java. At the moment of writing, only a beta version is available.

The two main concepts of JavaSpaces are 'space' and 'entry'. The term 'space' is used for the implementation of a JavaSpaces server. A space holds entries: typed groups of objects, expressed in a Java class. Three (atomic) operations are possible on entries:

- An entry may be *written* into a space. The result of this operation is a copy of the entry object, created in the space.
- An entry may be *read*, after which the space still holds the entry.
- An entry may be *taken*, after which the space no longer holds the entry.

With respect to the `read` and the `take` operations, JavaSpaces offers the possibility of matching against a template: entries in the space may be matched against an object that has some or all of its fields set to specific values (the template). The remaining fields act as wild-cards.

JavaSpaces offers still another kind of connection between clients of a space (or JavaSpaces server) and the space: clients may request a server to `notify` them, when an entry matching a specific template is written. Notification is done using a distributed event mechanism. This notify operation makes it possible to use the JavaSpaces package as an implementation of the subscription-based communication model.

Notifications may arrive in different orders on different clients, or may not arrive at all. The JavaSpaces server will make a 'best effort' attempt to deliver the notification. The server will retry at least until the notifications request 'lease' is expired. This 'lease' is an amount of time, negotiated between the server and the client, during the request for notification. This concept of a lease is added to the JavaSpaces package to obtain more robustness, in case of failure of parts of a distributed application.

Other implementations

A real-time publisher/subscriber model [Rajkumar and Gagliardi 1996] was developed at Carnegie-Mellon as a basis for building replaceable software units. The emphasis is on real-time, fault-tolerant systems. In addition to the basic publish/subscribe model, a facility is provided to detect whether nodes in the network have failed, and when new nodes in the network have joined. Therefore, channels are consistent to all publishers, i.e., all publishers on one channel will see the same subscribers at every time.

The Tibco Rendezvous Information Bus [TIBCO 1997] is a commercial implementation of the subscription-based communication model. It is mainly intended for financial applications, for example, the distribution of stock price information

to dealer rooms. One of the interesting features is that when a process subscribes, it can leave part of the channel name unspecified. The effect is that the process is subscribed to all channels matching the given channel name.

The Java Shared Data Toolkit [Burridge 1999], a product of Sun Microsystems, is intended for collaborative interactive applications. It provides the concept of sessions, which processes can join and leave dynamically. Within a session, processes can send messages to the other processes, and they have the ability to create shared data structures. The JSDT can be used on top of a number of different communication mechanisms, such as sockets or Remote Method Invocation. Sun Microsystems offers JSDT and JavaSpaces as separate products intended for use in different application areas.

8.2.3 A Java implementation of the radio broadcast paradigm

To have a facility for doing experiments with the RBP and an implementation, we developed a library implementing the Radio Broadcast Paradigm [de Rooij 1998]. Having a custom implementation allows us to experiment with different implementation techniques and to instrument the library in order to obtain performance figures.

The library provides an Application Programming Interface (API), containing two Java classes and one Java interface (Figure 8.2).

```
public class Transmitter {
  public Transmitter(String channel);
  public void transmit(Object object);
  public void close();
}

public class Receiver {
  public Receiver(String channel, Listener listener);
  public void close();
}

public interface Listener {
  public void accept(Object object);
}
```

Figure 8.2: Public classes and methods in Radio Broadcast Paradigm library

An application that wishes to open a channel to publish data, simply creates an object of class `Transmitter`, passing the name of the channel to the constructor.

To open a channel to receive objects, a process creates an object of the class `Receiver`. The constructor of the `Receiver` class takes the channel name and an object that implements the `Listener` interface as its parameters. This `Listener` object is responsible for dealing with objects received from the channel (the concept of a `Listener` object is similar to the GUI call-back mechanism introduced in JDK 1.1). The only method that must be implemented by a class that implements the `Listener` interface, is the `accept` method which is called whenever an object is received. It is up to the `Listener` object to handle the object.

The classes `Transmitter` and `Receiver` and the interface `Listener`, are sufficient to build complete applications. The programmer does not need to be not aware of the fact that `Transmitter` and `Receiver` objects may be on different machines.

The need to define one's own classes implementing the `Listener` interface for even the most basic ways of handling incoming data is rather impractical. Therefore, the library contains a few `Listener` classes for simple processing tasks. They are summarized in Table 8.1.

`StateListener`	keeps last received object
`QueueListener`	places received objects in a queue
`IndexListener`	places incoming objects in a table indexed by a key value; new objects overwrite older objects with the same key
`Filter`	forwards received object to another Listener only if a specified predicate is valid
`ClassListener`	if incoming object is a class description in Java byte code format, this class is defined and is linked dynamically with the Java virtual machine

Table 8.1: `Listener` classes

The behavior of `StateListeners`, `QueueListeners`, `IndexListeners` and `Filters` is fairly intuitive. The `ClassListener` class is more complex; it opens up the possibility of dynamically updating the code in a running system. New parts of the software can be distributed in the architecture-neutral Java byte code format to all processes that have a `ClassListener` defined for a certain channel. The class will be defined and linked dynamically with the Java Virtual Machine in the receiving process. If the class contains any static initializers, these will be evaluated, thereby allowing a class to initialize itself, for instance, by starting a new Thread. The `ClassListener` mechanism is similar to the on-demand loading of classes in Java-based web browsers. However, in our case, loading of classes is *push-based*. It is the transmitting side that decides if and when classes are to be sent. Since the usual Java dynamic class loading mechanisms are *pull-based*, missing class dependencies can be solved by loading the required class. In our case, missing class dependencies will cause the class defining and linking process to be suspended until the required class is received.

In the example given in Figure 8.3, we show how a `ClassListener` object is used to transfer actual byte code to another object after which the transferred code is executed. In this example, program `HelloProducer` constructs an object of class `ClassDefinition`, containing the byte code for class `HelloPrinter`. `HelloPrinter` is a class that contains a static initializer which will cause the string "Hello, World" to be printed on the screen. `HelloConsumer` is a simple program that only defines a `ClassListener` and then waits indefinitely.

On execution, the code for the class `HelloPrinter` is sent to the `ClassListener` that is contained in objects of the class `HelloConsumer`, and the text "Hello, World" will be printed on the screen.

```
public class HelloPrinter {
  static { System.out.println("Hello, world"); }
}

class HelloProducer {
  public static void main(String[] args) {
    try {
      Object o =
        new ClassDefinition(Class.forName("HelloPrinter"));
      new Transmitter("hello").transmit(o);
    } catch (java.io.IOException e) { }
  }
}

class HelloConsumer {
  public static void main(String[] args) {
    new Receiver("hello",new ClassListener());
    pause(); // Wait indefinitely
  }
}
```

Figure 8.3: A simple example in which Java byte code is transferred

Even though the `transmit` method takes an `Object` as parameter, only objects that directly or indirectly implement the interface `Serializable` can be transmitted. Objects not implementing this interface are silently discarded. The interface `Serializable` was introduced in Java 1.1 as part of the Object Serialization API. Our implementation of the Radio Broadcast Paradigm library uses these features internally to convert objects to byte arrays before passing them on to receivers. Even when sending objects within a single Java Virtual Machine, they are serialized and de-serialized.

Consistent serialization has the interesting consequence that any data structure that is transmitted will become a 'deep copy' when received. Consider for example a Vector of objects. Applying the `clone` method to such a Vector will yield a copy of the vector, however, any reference in the vector refers to the same objects as the references in the original vector. However, sending and receiving the vector through the Radio Broadcast Paradigm library will yield a vector with references to *copies* of the elements of the original vector. This is important when considering the fact that one of the purposes of the Radio Broadcast Paradigm is to decouple processes as much as possible from each other.

8.2.4 Design considerations

It is important to realize that the channels provided by the library are *unreliable*. Even though the library makes effort to get objects from transmitters to receivers, delivery of objects is not guaranteed. Similarly, the order in which objects are re-

ceived is not guaranteed. Even though two objects sent sequentially by the same transmitter will probably be received in the same order by all receivers, two objects sent by different transmitters may be received in different orders by different receivers. An application should not make any assumption about such issues.

Besides platform-dependent native-code solutions, the only form of communication between different Java virtual machines is the use of TCP/IP-based socket communication. Other high-level protocols, such as Java's Remote Method Invocation are built on top of this. Our implementation also uses TCP/IP sockets for communication between `Transmitters` and `Receivers` located in different Java virtual machines.

The implementation uses one 'server' process for each machine in the network that hosts processes using subscription-based communication. Even on a single non-networked host, this server process is still necessary to provide communication between processes running concurrently on this machine. Figure 8.4 illustrates the possible communication paths; the 'clients' in this figure are Java virtual machines that execute programs using the subscription facilities.

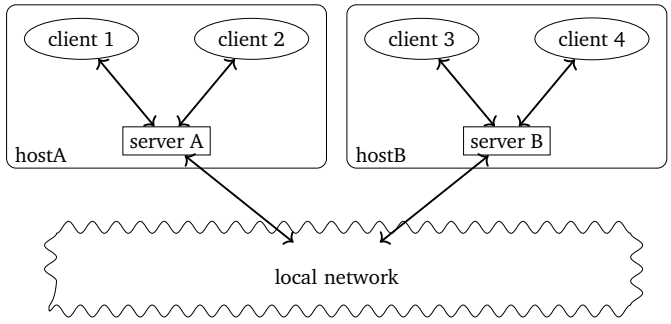


Figure 8.4: Communication between clients and servers

Servers periodically send broadcast messages on the network, allowing other servers to locate each other without using (static) configuration files. Broadcasting furthermore allows servers to detect when other servers have crashed (to be deduced from the absence of messages for a longer period of time).

For exchanging actual data over channels, server-to-server UDP messages are sent. The use of UDP, a datagram-oriented protocol, rather than the reliable TCP protocol, is based on the assumption that most embedded systems deal with a continuous stream of data from the environment, and that the environment serves as a back-up for occasionally missing a data item [Boasson 1996]. Since TCP is a reliable stream-oriented protocol, the operating system will guarantee that all messages are delivered in the right order. If TCP was used, a packet dropped by the network would cause the data stream to stall until the dropped packet would have been been re-sent.

Clients communicate with servers using TCP rather than UDP, since using UDP would have no advantage here. Communication between processes on a single host

is always reliable, so the previous scenario in which a missed message causes a stall will not occur.

Periodically, the servers communicate with each other and with the clients attached to them, through the set of channel names for which their clients have defined `Receiver` objects. Availability of this information minimizes the amount of data exchanged between servers and clients, and among servers. Consider, for instance, the case in which two threads within the same process communicate with each other. If there are no other processes that have defined `Transmitters` or `Receivers` with the same channel name, the objects communicated between the two threads will not be sent to the server. However, as soon as somewhere in the network a `Receiver` is defined for that channel, the server to which it is attached will announce this to the other servers, and future objects sent on the channel will be sent to the new receiver as well.

An object diagram of the client-side library of the Radio Broadcast Paradigm implementation is shown in Figure 8.5. In this figure we show the `Transmitter`, `Receiver` and `Listener` objects. The `Transmitter` and `Receiver` objects contain a reference to the `Channel` object corresponding to the channel they are attached to.

For each channel name, a `Channel` object exists. The mapping between channel names and `Channel` objects is maintained by a (single) `Registry` object. Finally, the `Forwarder` class, which is also a singleton, is responsible for communication with the server. It multiplexes and de-multiplexes incoming and outgoing data and control messages to a single TCP/IP connection.

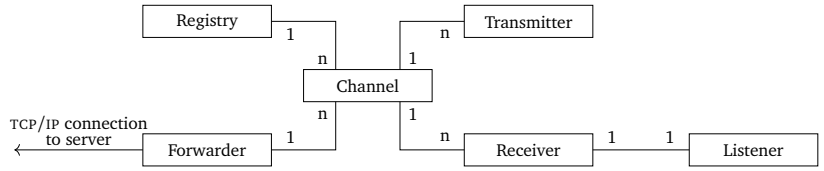


Figure 8.5: Objects in the Radio Broadcast Paradigm

The server program contains a single 'manager' object, and two objects for each connection to the server. These objects handle the incoming and outgoing data stream, respectively. The manager object keeps a table with information about which channels should be forwarded to which connections. In addition, there is a separate thread that periodically broadcasts to the network to discover the presence of other servers on the network. If a new server is found, a new connection is set up.

The use of Java provides a high degree of portability.

8.3 Controlling through subscription: A case study

8.3.1 Problem description

Rotterdam is the world's largest harbor. One of the largest container terminals in Rotterdam's harbor is fully automated. Cranes are used to carry the containers from a ship onto a lorry or from a lorry to a ship, from lorry onto stacks or onto a train or from trains or stacks onto a lorry.

The terminal employs a system of unmanned, automated vehicles transporting containers. The current system is running for several years now, and it is near its limitations. In a next-generation system, the number of lorries that can be handled has to increase dramatically, while the system should furthermore be able to handle lorries with different characteristics. Finally, traffic will be extended to include drives between terminals. There is no doubt that the complexity of the system will increase dramatically.

In order to get a good understanding of the problems involved in controlling the handling of these large amounts of containers, a prototype simulator/controller is being developed that is being used as a test-bed for a variety of implementation issues. The test-bed is based on control being distributed: each vehicle has its own controller, dictating its behavior. The main reason for this design choice is that a distribution model conceptually fits perfectly to the situation of unmanned vehicles moving around. In practice, we will have to evaluate performance before deciding whether central or distributed control is more effective. Our implementation of the test-bed is using the earlier mentioned implementation of the RBP. The approach taken in our study was motivated by our cooperation with another group that is building a similar prototype using a centralized control approach, using 'hierarchical semaphores' [Evers 1999] as controlling devices.

8.3.2 Requirements and design decisions

The current lorry handling system is able to handle a few dozen cranes and lorries. The system modeled through our test-bed should be usable for larger systems, systems with, for instance, 1000 lorries rather than the current 50.

The new system will have to handle changes in the number of lorries that are being dealt with during operation. However, the system also has to take into account the addition of lorries with different kinds of behavior. Main requirements for a solution are therefore:

scalability with respect to the number of lorries and the number of movements,

flexibility with respect to changes in the number of lorries and the behavioral patterns of the various vehicles operating on the terrain.

For this article, it is assumed that a separate planner system exists that provides each vehicle with a plan: a function mapping time to the position the vehicle should be (or should have been) at that moment. A complete plan provides information on

where to collect the container, how to drive through the area and where to deliver the container.

In the design of a system based on an RPB, one must keep in mind that there is no guarantee that data will arrive. The only countermeasure is to send precious data twice or more times. However, in a case like this, the loss of a single message is no disaster: the receiving processes temporarily use information slightly older than it should be.

8.3.3 Outline of the control system

We discussed the control system we designed for our case study in detail [Stuurman and van Katwijk 1998] (this article is used for chapter 4). Here, we briefly outline the ideas and the structure of the implementation.

The overall guiding principle is separation of concerns. Separation of concerns on control is obtained by modeling a controlling process for each vehicle in the area, and by letting each controlling process be responsible for the control decisions on the movements of the associated vehicle. To do so, each vehicle controlling process needs a consistent view on the area it is in.

The system is built such that a vehicle control process can deduce a sequence of places that it will pass from its given plan. Essential in the model is that the process knows the planned position for the vehicle at any time. Furthermore, it knows the execution attributes of the vehicle it controls, such as the actual position, the velocity and the driving characteristics.

Vehicle control processes send their short-term plans (the part of the detail plan that should be followed in the immediate future) periodically through a channel. Vehicle control processes furthermore listen to the short-term plans of other vehicles and evaluate possible collisions. Each vehicle obeys a set of traffic rules, and knows the traffic rules of all other vehicles. Traffic rules can be as simple as

- *The lorry with the lowest order number has precedence* or
- *The lorry that is in the greatest hurry has precedence*

or as complex as

- *A vehicle coming from right has precedence, unless the vehicle from left is delayed more than the vehicle coming from right.*

Under the assumption that any vehicle has a consistent view on the area, application of valid traffic rules allows a control process to take a decision on whether to continue, to stop or to alter the velocity in the presence of other vehicles.

We realize that in order for a lorry to take a decision, having a consistent view on the *whole* area is an overshoot. What is required is a consistent view of the direct environment of the lorry. Therefore, the area in which the vehicles move is partitioned in regions: a grid of for instance 10 by 10 coordinated points. To each region one particular channel is associated. A vehicle sends its short-term plan through the channel, associated with the region it is positioned in; it listens to the

same region channel, and to the channels of the eight regions surrounding this region as well. Scalability is promoted through this approach.

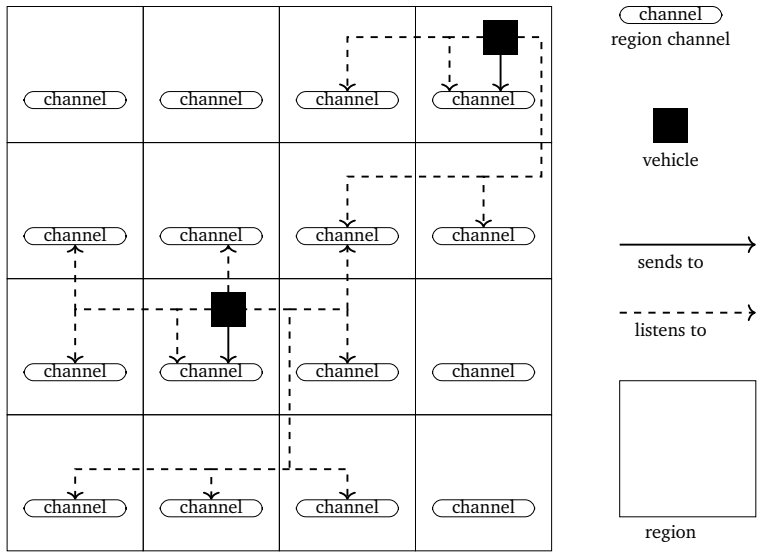


Figure 8.6: Vehicles in regions

In Figure 8.6, we show two vehicles in two different regions. Each vehicle sends (solid line) to the channel associated to the region it is positioned in. All vehicles in a given region listen to the same channel, and to the channels associated to the eight regions around it. Note that it is possible that more than just one vehicle drives around in one region.

Furthermore, a region process that collects data of the vehicles driving in the region, is associated to each region. It creates a summary which is sent through the 'image channel', available for our visualization system. The visualizer can also be used to zoom in to one region, by telling it to listen to the associated channel. The short-term plans of the vehicles in that region may be inspected.

The traffic rules are a part of each vehicle process. Every vehicle process listens to a 'rules' channel, which may be used to send a new version of these traffic rules. To avoid version conflicts, vehicles send the version number of the rules they use, together with their short-term plan. In case of a version conflict, a default rule is used by both parties.

8.3.4 Implementation

The controller is built using the RBP library that was discussed in Section 8.2.3. In Figure 8.7 we show the internal view of a vehicle process. In this figure, active (threaded) objects are represented by a rounded box, while sharp-edged boxes denote threadless objects.

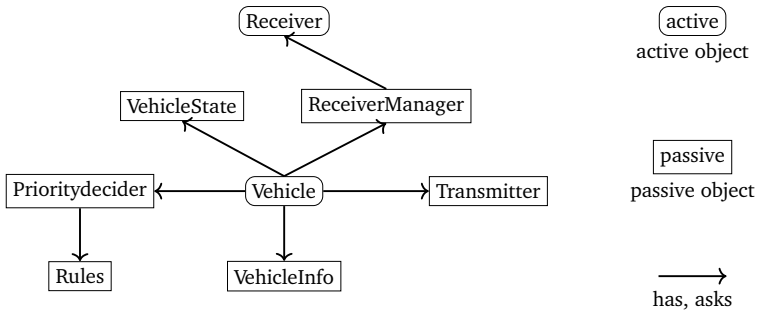


Figure 8.7: Objects in the vehicle process

The vehicle process performs a loop:

- It asks the `VehicleState` object for the speed and position of the vehicle, the short-term plan and the current version of the traffic rules, and adds its identity to construct the current `Vehiclelnfo`.
- It asks the `Transmitter` to transmit the `VehicleInfo` object.
- The `Receiver` objects, which are connected to the `ReceiverManager` object, listen for information from other vehicles during the loop.
  The `ReceiverManager` object is responsible for the choice of the channels to listen to. It selects these channels based on the position of the vehicle.
- Once every few time units, the `PriorityDecider` is asked to compute the next position the vehicle will drive to. This is based on the information gathered from the local environment and – obviously – on the traffic rules. The object `VehicleState` is updated.
- The traffic rules themselves may be updated when a new class definition is sent through the `Rules` channel.

8.4 Analysis of the case study

8.4.1 Our approach to abstraction and verification

As stated in the introduction, analysis and verification should provide feedback on the design and implementation of real-time distributed control systems. Our focus is on analyzing applications and deriving bounds for the execution parameters of the underlying system, in our case, the RBP implementation. The technique we use for this kind of analysis is model checking.

Model checking is becoming more and more part of the development process for hardware and software products. Companies like Intel and Lucent are using model-checking in their development processes and are even developing their own model-checkers. Model-checkers customized for hardware design are becoming commercially available [Kurshan 1997]. Integration of model-checking in software development processes is not yet that far developed. There is, however, an increasing number of successful applications within the software industry, but it will take some time before it matches with the applications in the hardware sector.

The computational model that is generally used in the software case is (a variant of) the finite state automaton. The basic model of a finite state automaton provides only a basis for control abstraction. As the model became more widely used, constructions were added that enable the specification of data values and data transformations. To be of use in the field of real-time systems, time and synchronization were added.

We developed a notation, eXtended Timed Graphs [Ammerlaan et al. 1998], an automaton-based formalism [Alur and Dill 1994] with extensions for data specification and a general form of control semantics. The primary use of the notation is as input notation for our model-checker.

The XTG notation is *not* meant to be a modeling language in itself. Other, better suited, modeling languages exist that focus more strongly on the various aspects of system and software design, such as composition and decomposition, reuse, object-orientation, et cetera.

XTG is designed particularly to use as a verification language. It is supported by a more human-oriented specification language ATL, the Abstract Thread Language, which is currently developed by our team. An ATL specification can be translated into XTG's. Such XTG's can be fed into the model-checker. In our approach, designs and implementations are abstracted into ATL specifications, that are fed into the model checker.

Syntax and semantics of the behavioral constructions in ATL are based on C and Java. Furthermore, ATL supports a simple communications model which is loosely based on the subscription paradigm [Boasson 1996; de Rooij 1998]. Communication takes place along 'channels'. To send information on a channel, the send-operation is used, with the syntax $channelname.send(value)$. To receive information from a channel, one defines a receiver thread. A receiver thread declaration consists of the receiver name, the variable that is used to store the value received from the channel, and a statement sequence to be executed every time the send-operation is invoked by some thread. A send-operation will not block until the receiver thread is ready

to accept the communicated value. If a receiver thread is not idling, the value to be sent by the send-statement will be lost.

XTG and ATL share a single property specification language, a temporal logic based on CTL (Computation Tree Logic) [Clarke et al. 1986] and TCTL (Timed CTL) [Henzinger et al. 1994]. The usage of these logics is strongly connected with the application of model checking verification. TCTL variants are real-time extensions of CTL. These extensions either augment temporal operators with time bounds, or use reset quantifiers.

The core syntax of CTL defines two temporal operators, `AU` and `EU`. The formula $(\phi_1 AU \phi_2)$ is satisfied in a state if for all computation paths starting from that state, a state that satisfies ϕ_2 is encountered, and until that time ϕ_1 is satisfied. $(\phi_1 EU \phi_2)$ is satisfied if there is at least one such computation path. Derived operators are: $EF\phi$ (There is path on which there is state satisfying ϕ), $EG\phi$ (There is a path on which every state satisfies ϕ), $AF\phi$ (On all paths there is some state satisfying ϕ), and $AG\phi$ (On all paths every state satisfies ϕ).

Our CTL variant supports two types of atomic properties: boolean expressions over values of variables and clocks of both the system and the property, and location expressions. The latter take the form $g@l$, which expresses the fact that the graph g of the system is currently at location l.

To address resetting quantifiers, we use a technique found in other TCLTs, which consists of modeling the reset quantifier by assignments to specification clocks and variables. As an example, in our CTL variant (referred to as CTL_{XTG} or simply CTL),

$$z := 0.AF(p \wedge z \leq 10)$$

expresses that p will always become true sometime within ten time-units. A secondary benefit from this approach is that we can use symbolic constants in our property formulae (as suggested by Henzinger [Henzinger 1996]). Consider for example, the CTL formula

$$AG(t := count.AF(count := t + 1))$$

in which $count$ is a system variable and t is property specification variable.

8.4.2 A model of the controller

The correctness of operation of the controller we discussed in Section 8.3.3 does also depend on the behavior of the underlying RBP implementation. As discussed in Section 8.1, we therefore should model the behavior of the controller implementation, identify the dependencies on the underlying implementation, and derive bounds for the various parameter values. In this section, we demonstrate the usefulness of such an analysis using a highly simplified version of the autonomous vehicle system as described in Section 8.3. In the simplified scenario discussed here, we take only two vehicles into account. These vehicles drive in a straight line towards each other, as shown schematically in Figure 8.8.

Each vehicle stores two positions, its own and the position of the other vehicle, both represented by a simple integer value. It is assumed that the vehicle's own

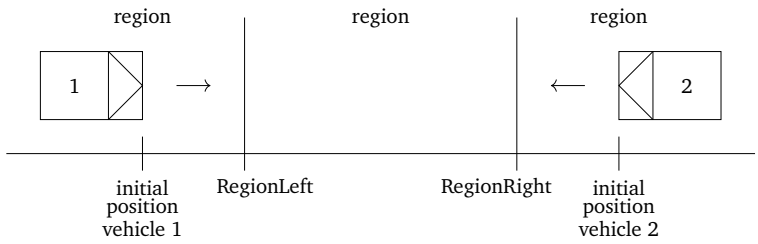


Figure 8.8: Scenario of the simplified model

position is accurate, but the position it believes the other vehicle has, is based on the information it received.

Even though this model is an oversimplification of the original system, it contains the essential elements: vehicles publishing their own position, and making decisions based on received information. The communication model is unreliable, in the sense that a message will be lost when a receiver is not listening, for instance, because it is busy processing a previous message.

In the ATL specification each vehicle is modeled by a controlling process and a receiver process. The controlling process periodically sends the vehicle's position to the receiver process of the other vehicle. The controlling process of a vehicle retrieves the (assumed) position of the other vehicle from a variable shared with the receiver process. Figure 8.9 shows the code of the controlling process and the receiver of one of the vehicles. The code for the other vehicle (defining the position variable $p2$) is similar and is omitted here.

```
// Vehicle1's position
int p1 = Vehicle1Initial;
// Vehicle1's idea of Vehicle2's position
int q1 = Vehicle2Initial;
// Controlling process for Vehicle1
process Vehicle1 {
   forever {
      if (p1 >= RegionLeft && p1 <= RegionRight)
         plan1.send(p1+1);
         sleep(VehicleDelay);
      if ((q1-p1) > Distance || (p1-q1) > Distance)
         p1++;
   }
}
// Receiver process in vehicle 1
// receives position from Vehicle2
receiver plan2(int buffer) {
   sleep(ReceiverDelay);
   q1 = buffer;
}
```

Figure 8.9: ATL code of the vehicle

Vehicle1Initial	0
RegionLeft	3
RegionRight	12
Vehicle2Initial	15
Distance	2
VehicleDelay	4
ReceiverDelay	7

Figure 8.10: Constants used in experiments

8.4.3 Analysis of the model

In our experiments we are interested in the the safety property 'the two vehicles will never collide' and under which parameter values, i.e. indicating execution attributes of the underlying RBP implementation, the property holds. The non-collision property is expressed as $AG\ p1! = p2$. A vehicle might collide with the other one if the perceived position of the other vehicle differs sufficiently from the actual position of that vehicle. Such an inconsistent view might be obtained when processing of the received positions is too slow. Several verification experiments were performed based on the above model.

Verification of the non-collision property In a first experiment, we verified the safety property for a given set of constant values. The chosen values for the constants are shown in Table 8.10. As can be derived from the ATL description of the vehicle, `Vehicle1Initial` and `Vehicle2Initial` indicate the initial positions of the vehicles. `RegionLeft` and `RegionRight` indicate the region, `Distance` indicates the required minimum distance between the two vehicles, `VehicleDelay` indicates the time between two iterations of the `Vehicle` processes and `ReceiverDelay` indicates the Receiver delay. With these values, the safety property holds.

Deriving the communications region In a second experiment we derived the smallest communications region possible for which no collision will occur. When the region is too small, the vehicles will not communicate their position fast enough to each other to allow a reaction. If `RegionLeft` is made a parameter, the safety property is shown to be satisfied when `RegionLeft` ≤ 6. In the same way, the bound `RegionRight` ≥ 9 was derived.

Deriving upper-bounds for parameter values A third experiment was aimed to derive bounds on the various delays in the vehicle program. First, we determined an upper bound on the `ReceiverDelay`. Too high a value for the `ReceiverDelay` will cause the vehicles to collide, as they will not have a correct idea of the position of the other vehicle. Except for the value of `ReceiverDelay`, we used the same constant values as in Table 8.10. The safety property is satisfied for `ReceiverDelay` ≤ 7.

A similar (fourth) experiment was performed for the value of `VehicleDelay`. To ensure that the verification process ended, we needed to impose an upper-bound on the value of `VehicleDelay`, by replacing the statement

```
delay(VehicleDelay);
```

by

```
if (VehicleDelay < VehicleDelayMax)
    delay(VehicleDelay);
else
    delay(VehicleDelayMax);
```

where `VehicleDelayMax` is an appropriately chosen constant. After this modification, the property is satisfied for `VehicleDelay` ≥ 4.

Deriving dependencies between parameters In the fifth experiment, we took two parameters, `VehicleDelay` and `ReceiverDelay`. The values, derived by our model checker PMC, for which the safety property is satisfied are shown as the shaded area in Figure 8.11.

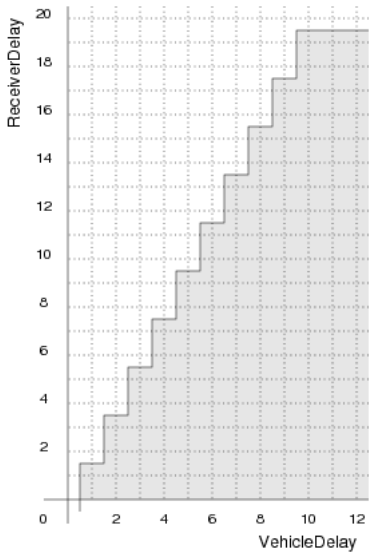


Figure 8.11: Verification result using `VehicleDelay` and `ReceiverDelay` as parameter; shaded area shows where the safety property is satisfied

In this experiment, we limited the values of `VehicleDelay` to the range from 0 to the value of `VehicleDelayMax` of 10. The relationship between `VehicleDelay` and

`ReceiverDelay` approximates a linear relation described by:
`ReceiverDelay` $< 2 \times$ `VehicleDelay`.

As the delay value in the vehicle controlling process gets longer, the communications layer is allowed to have a longer latency. The coefficient of 2 can be explained by the value of `Distance` (the minimal distance vehicles will try to keep between them). Changing the value of `Distance`, changes the coefficient as well.

Finally (sixth experiment), we considered independent delays of both receivers In Figure 8.12.

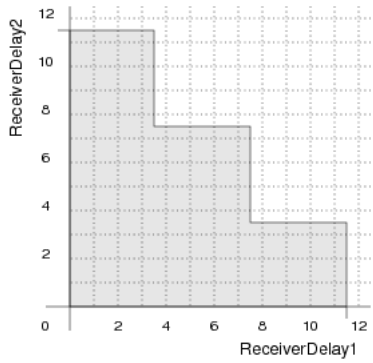


Figure 8.12: Verification result using `ReceiverDelay1` and `ReceiverDelay2` as parameter; shaded area shows where the safety property is satisfied.

We show the values of `ReceiverDelay1` and `ReceiverDelay2` for which the vehicles will not collide. From this figure, it can be seen that a longer communications latency in one direction can be compensated by a shorter delay in the other direction. This result shows a relationship that cannot be described by an approximation of a linear relation. This can be explained by realizing that the vehicle controller process 'samples' the position of the other vehicle at regular intervals of `VehicleDelay`. Changing the value of `VehicleDelay` will change the size of the 'blocks' in the figure, although the shape will be similar.

Results

The results of the verification experiments described above are shown in Figure 8.13, where `p1` and `p2` are positions. The objective of these experiments is to demonstrate the use of our tool-set for the (formal) derivation of values for which the required property holds. The parameter values themselves denote execution attributes of the underlying system. The result indicates the ranges for the values of the various attributes such that the property is guaranteed to hold.

#	*Property*	*Parameters*	*Result*
1	AG p1 != p2	—	positive
2	AG p1 != p2	`RegionLeft`	`RegionLeft` ≤ 6
	AG p1 != p2	`RegionRight`	`RegionRight` ≥ 9
3	AG p1 != p2	`ReceiverDelay`	`ReceiverDelay` ≤ 7
4	AG p1 != p2	`VehicleDelay`	`VehicleDelay` ≥ 4
5	AG p1 != p2	`ReceiverDelay`, `VehicleDelay`	Figure 8.11
6	AG p1 != p2	`ReceiverDelay1`, `ReceiverDelay 2`	Figure 8.12

Figure 8.13: Verification results

8.4.4 Running the model

The next step is to investigate under which conditions the underlying system has values within the ranges specified above as its execution attributes. To get some feeling of the results obtained analytically, we performed a series of experiments with the same system that was used as a basis for analysis. The system simulated (and controlled) the behavior of two vehicles.

One vehicle had as its mission to drive from point (0, 0) to point (30, 0); the other had to drive the opposite way, both with a step-size 1. The experiments were carried out three times:

- once with both vehicles running on a Linux system,
- once with both vehicles running on a Windows NT system,
- once with one vehicle running on the Linux system, and the other on the NT system.

In all cases the simulation was checked for collision between the vehicles.

Varied parameter : The parameter that was varied in the experiment, was the cycle-time of the vehicles. This cycle-time determines the `VehicleDelay`, which amounts to the cycle-time plus 5 milliseconds.

Measured parameter : The parameter that was measured was `ReceiverDelay` (the time between the moment a message is sent, and the moment it is received), in relation to the occurrence of collisions.

By default, the cycle-time of the vehicles was set to one second, meaning that each second, a vehicle checks the data it did receive giving information on the position of the other vehicle, and it computes its next step based on those data. Measurement shows that the time needed for the computation is approximately 5 milliseconds (on both systems). The actual cycle-time is therefore 1005 milliseconds; the measured

`ReceiverDelay` varied between 0 and 12 milliseconds, with consistent results in all three experiments. With these values for `CycleTime` and `ReceiverDelay`, simulation showed that the vehicles always detect each other in time, and no collision occurs.

Varying the cycle time gives consistent results. The measured `ReceiverDelay` varies between the same values, whether both vehicles are run on the Linux system, on the NT system, or both on a different machine. The results are presented in Figure 8.14, in which the unit of time is milliseconds.

CycleTime	*ReceiverDelay*	*Detect on Time*
1000	0-12	yes
5	5400-6600	no
100	2900-3800	no
300	500-600	sometimes
350	250-360	yes
500	170-370	yes

Figure 8.14: Results of the experiment

It shows that a shorter `CycleTime` results in a longer `ReceiverDelay`. For our network (where the `ReceiverDelay` does not depend on whether the vehicles run on different systems or on the same system), the critical value of the `CycleTime` appears to be about 300 milliseconds: as long as the value of `CycleTime` is larger, collisions are avoided. When shorter `CycleTime`s are chosen, collisions almost certainly will take place.

These results are consistent with what was derived from the results of the verification experiment, summarized in Figure 8.11. When `ReceiverDelay` becomes larger than 8 (twice 4), collisions cannot be avoided.

8.5 Conclusions and further work

In this study we introduced our approach to software development of dynamic real-time distributed systems, based on complementary use of experimentation, abstraction and verification. The focus of our study was on a particular approach to model the middleware between distributed control applications and the underlying network structure, the Radio Broadcast Paradigm (RBP).

We created a test-bed for analyzing applications using our RBP through a Java implementation and experimented the RBP on an industrial case study. Abstraction of the Java implementation into a formal abstract language and the translation of the resulting representation into a modeling language for parametric verification, enabled the derivation of constraints on the underlying execution environment. Our combined approach thus provides a sophisticated framework for the development of evolving distributed real-time systems.

Our experiments show that the approach is viable; the results of our experiments are very encouraging, although scaling is an issue. When more vehicles are involved,

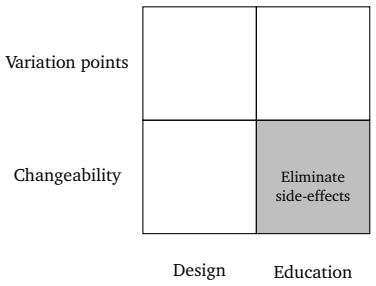
experiments based on model-checker predictions are more complex and the model-checking itself requires far more resources. One practical issue to be addressed is visualization of results: as more variables depend on each other, visual representations of the dependencies become complex.

Current validation runs were done on a large Linux machine, but could have been done on any PC. Our current experiments involve larger control configurations (over a dozen vehicles involved) than given here, and verification runs require somewhat more memory than available on an ordinary PC. However, due to the symmetry in the ATL specification, the amount of memory for model-checking in the complete verification remains within limits. The models provide useful insight in further optimizations in the model checker.

Our current work consists of optimizing the model checker, and developing robust translators for notations like UML, Java and C languages into ATL specifications.

9

Creating a short course on Scala[1]



In this article, we show how we created a short on-line course on the programming language Scala, as a side-product of our own professional development. By developing this course, we allow students to take advantage of the combination of functional programming (with the advantage of elimination of side-effects) and object-oriented programming, without having to change the current curriculum.

Relevance to this thesis

With this short course, we contribute to the subject of education within the field of design for change, by enabling developers with experience in object-oriented languages to learn to use a functional style, creating functions without side-effects.

Another link to this thesis is the fact that by creating short courses, we enhance the changeability of our curriculum: it is easy to adopt parts of short courses to use in regular courses, or in new courses that wil become part of the curriculum.

Deviations from the original article

The main deviation is that we changed the order of the sections.

[1]This article originally appeared under the title of 'A New Method for Sustainable Development of Open Educational Resources' in the Proceedings of the Second Computer Science Education Research Conference CSERC, pages 57-66, 2012.

Abstract

Open Educational Resources (OER) seem to be a natural fit with a distance learning university: open resources are in line with the university's mission to provide access to academic education, material is often available in digital form, and even the name of distance learning universities often contains the word 'open'. However, in practice, it is difficult to realize sustainable OER, especially if existing material may not be used. We propose a new method to create sustainable OER, based on new educational material, and compare this method with existing models for sustainable OER. The main characteristic of the method is that OER are produced as a side-effect of Continuous Professional Development (CPD). As an example of our CPD method, we describe the development of a short OER course about the programming language Scala.

9.1 Introduction

For some universities, it is too big a risk to offer all educational material for free in the form of Open Educational Resources (OER). This holds, for instance, for a distance learning university that prepares its own education material, complete with guidelines, exercises, and other elements that enable students to follow the course (in principle) without any further guidance[2]. The risk of losing considerable income when offering such material for free is very real.

On the other hand, distance learning universities and OER seem to be a natural pair. The mission of a distance learning university is to enable everyone to enjoy academic education, to stimulate students to educate themselves at an academic level, and offer them the means to do so. Many distance universities – ours is an example – have the word 'open' in their name. In other words, the desire to contribute to OER is great.

Within our university, there has been an effort to prepare short courses, for free, in the form of OER [Schuwer and Mulder 2009]. This project was funded both by the Dutch government and the William and Flora Hewlett Foundation, and has now ended. A sustainable method for creating OER cannot rely fully on such funding.

The contribution of this article is that we propose a method for sustainable development of OER, in the form of short courses, not based on existing material but on new material, almost without extra funding. The core of the method is to produce OER as a side-effect of Continuous Professional Development (CPD) for teachers [Day 1999]. For that reason, we coin our method *the* CPD *method*.

We describe the development process of the CPD method and argue why this process is sustainable. We propose requirements for a suitable course subject, and we present the results of a course, prepared using this method. We compare our approach with existing models of sustainable OER, and discuss advantages and disadvantages of our approach.

In this article, we address the following questions:

[2]The Open University of the Netherlands is such a distance learning university.

- What are the characteristics of the CPD method to develop an OER course, and how does it differ from existing models of sustainable OER development?
- Is it possible to engage students and non-students during the development of a course, and how can they participate? What works, what does not work, in that respect?
- What are the criteria for the coice of a subject to be used in this type of OER development?
- What are the advantages and disadvantages of the CPD method and the resulting type of OER?

The remainder of this article is structured as follows. We start by introducing Open Educational Resources and existing models for sustainable OER in Section 9.2. In Section 9.3, we introduce an example OER product which is developed with the CPD method: a short course on Scala, which functions as a running example illustrating our approach. Next, we describe the development process (Section 9.4) of this first short course. In Section 9.5, we discuss the characteristics of our CPD method, with respect to the aspects of funding, technical, and course content. In this section, we also elaborate on the differences with the existing models. Section 9.6 then discusses how the subject of this course meets the criteria of the CPD method. We conclude the article, in Section 9.7, with an evaluation of the feedback from our readers, and draw some conclusions and discuss alternatives.

9.2 Open Educational Resources

There are many names for educational material that is offered for free. 'Open courseware', 'open academic resources', and 'open educational resources' are among those names. The emphasis differs slightly: courseware, academic, educational. In this article, we will use the name 'Open Educational Resources' (OER), by which we mean material that can be used in education, and is made available to the public for free.

The idea of OER started as 'Open CourseWare' (OCW). The OCW movement advocates to offer the material used by universities to educate their students for free to the rest of the world [Caswell et al. 2008]. The original idea was that the material is available anyhow and that reproduction cost in the internet age is almost zero. Therefore, universities could, in theory, offer their material without additional cost. Other forms of OER are parts from lectures in audio or video, such as a demonstration of an algorithm or physical principle, or the recording of a complete lecture. According to Hylén [Hylén 2006], OER may contain:

Learning content: Full courses, courseware, content modules, learning objects, collections and journals. Learning objects have many different definitions; one of those definitions is 'any reusable digital resource that is encapsulated in a lesson or assemblage of lessons grouped in units, modules, courses, and even programmes. A lesson can be defined as a piece of instruction, normally including a learning purpose or purposes' [McGreal 2004].

Tools: Software to support the development, use, re-use and delivery of learning content, including searching and organization of content, content and learning management systems, content development tools, and on-line learning communities.

Implementation Resources: Intellectual property licenses to promote open publishing of materials, design principles of best practice, and localization of content.

In practice, the costs of OER are far from zero [Downes 2007]. The William and Flora Hewlett Foundation[3] offers a grant for universities and other parties to open their content, and many universities have funded OER projects themselves. In the long term, developing and maintaining OER should be possible without additional funding, in order to make it sustainable. The fact that the costs of OER are not zero does not only stem from the fact that it costs time (and therefore money) to prepare existing educational resources for publication on the internet. There is also a risk involved for distance learning universities that create their own educational material: it is not known how many students only buy courses to acquire the course materials, without the intention to complete the course. In general, the course material contains everything needed to study the course. Therefore, it cannot be foreseen whether offering material for free will increase income (because more people get acquainted with the university), or decrease income (because less students buy courses since the course material is free).

Hylén [Hylén 2006] lists some arguments for institutional involvement in OER. First of all, in line with academic traditions, one considers sharing knowledge as a good thing in itself. Second, educational institutions are funded (partly) from taxpayers' money, so it seems just to share and reuse resources developed by publicly funded institutions. Also, by reusing shared resources, the costs for content development can be cut, thereby making better use of available resources. Finally, one needs to look for new ways of making revenue, for instance by offering content for free both with the purpose of advertising the quality of the teaching institute and as a way of lowering the threshold for new students.

9.2.1 Sustainable OER

Sustainability of OER can be defined as follows: 'Having a mechanism in place for generating, or gaining access to, the economic resources necessary to keep the intellectual property or the service available on an ongoing basis' [Guthrie et al. 2008]. This definition is in line with the definition of sustainability of OER given by Downes: 'Having long-term viability for all concerned – meets provider objectives for scale, quality, production cost, margins and return on investment' [Downes 2007].

Two aspects can be discerned in these definitions. One aspect is the fact that an institution needs money to make resources available. There are costs involved in the production of OER, even if the resources already exist. It is this aspect that is our

[3] http://www.hewlett.org/programs/education-program

main focus. Another aspect is the fact that some resources need a mechanism to stay available on an ongoing basis. An example of such an OER is the Stanford Encyclopedia of Philosophy[4]. This encyclopedia clearly needs to be updated on an ongoing basis, adding new philosophers or new insights. This requires some mechanism to be in place. In this article we do not consider OER that require such a mechanism.

Downes [Downes 2007] describes funding models, technical models, content models, and staffing models for sustainable OER. We will provide a brief overview of these models.

Funding models

Since there are costs to make existing or new education material publicly available in the form of OER, sustainable OER needs to address how it is funded. Below, we sum up existing funding models.

Endowment Model: The project is sustained from interest earned on a fund. The Stanford Encyclopedia of Philosophy is an example of this model.

Membership Model: Contributions are made by interested organisations, e.g. the Sakai Foundation[5].

Donations Model: Donations are managed by a non-profit foundation. Wikipedia[6] uses this model of donations.

Conversion Model: Something is given away for free with the possibility to pay for extras. Several Linux distributors have adopted this model.

Contributor-Pay Model: The author pays (once) to the provider, who makes the contribution available for free. The Public Library of Science[7] is an example.

The Sponsorship Model: Sponsoring may have the form of advertisements or just mentioning the name. Examples are the MIT iCampus Outreach[8] (Microsoft) and the Stanford iTunes project[9] (Apple).

Institutional Model: An institution itself may pay for an OER initiative, such as MIT does for its OpenCourseWare project[10].

The Governmental Model: Here, the governmental model represents direct funding for OER projects. An example is the Dutch Wikiwijs[11] project.

[4] http://plato.stanford.edu/
[5] http://sakaiproject.org/sakai-foundation
[6] http://en.wikipedia.org/wiki/Wikipedia:Contact_us/Donations
[7] http://www.plos.org/publish/pricing-policy/publication-fees/
[8] http://icampus.mit.edu/outreach/
[9] http://itunes.stanford.edu/overview.html
[10] http://ocw.mit.edu
[11] http://www.wikiwijs.nl/home/

Partnerships and Exchanges: Partnerships depend not so much on exchanges of funding as on exchanges of resources, where the output of the exchange is an OER.

The Ithaka report [Guthrie et al. 2008] on sustainability of online academic resources distinguishes between funding by direct beneficiaries (such as subscription payment, one-time payment, pay-per-use, and contributor payment) and funding by indirect beneficiaries (such as host institutional funds, corporate sponsorships, advertisers, philanthropic funding, and licensing).

These models for financial sustainability all concern funding for the costs of OER. Another relevant aspect, which is not covered by these models, is a mechanism to *reduce* the cost of producing and maintaining OER.

Technical models

Technical models address how OER is made available, and how it is used. Roughly, Downes discerns two models:

- the OERs are used 'as is' without modification, or
- resources are downloaded, adapted, and sent back to the system repository for vetting and potential use by others.

He does not elaborate on the implementation, i.e. the platform needed for these two models. Obviously, a platform which allows for downloads of content only supports the first model, whereas Wikis or other community platforms allow for the second model as well.

Content models

Concerning content, Downes states that sustainability means that the content should be reusable: it should be possible to integrate content in another context. Our material is reusable by definition, in that sense: the material is designed to be studied without any extra guidance (although we do give extra guidance). Another content-related aspect is the licensing model used. Also, sustainable OER might need a community around it. This last point is in line with the recommendations of the Ithaka report [Guthrie et al. 2008]:

- understanding user needs is paramount but often neglected,
- create a competitive advantage, and
- catalysing a dynamic environment for agility, creativity, risk taking, and innovation is imperative.

Often, it seems to be taken for granted that OER consists of existing material, but this is not always the case. Wikipedia is an example of an OER for which new content is created, and in the OpenER project [Schuwer and Mulder 2009] of the Open University of the Netherlands, some of the OER courses were created using existing material, but other courses were created from scratch.

Staffing models

Staffing models concern the people involved in producing OER, and making material available. Downes mentions:

Producer-consumer model: OER is produced by professional staff. There is control over quality and content, but it requires great levels of funding.

Co-producer model: The consumers of the resources take an active hand in the production, or the production is done by volunteers. There is little control over quality and content, but this model requires much less funding.

9.2.2 Research on sustainability of OER

As is stated by Friesen [Friesen 2009], sustainablility is structurally excluded from surveys and other forms of research on OER. The Unesco report on OER [d'Antoni 2008] mentions the following three issues concerning sustainability: awareness rising and promotion, communities and networking of creators and users, capacity development.

Friesen [Friesen 2009] found that these issues are not solved for most OER initiatives. One of the exceptions is MIT Open Courseware [12]. The material is used all over the world, in the form of courses, and MIT benefits from the OER in several ways: it helps students and teachers at MIT itself, it increases the recognition of MIT as a leader in de subjects of the courses, it benefits the recruitment of students. Friesen concludes that similar initiatives in OER will be expected from other universities, in the future. In other words, in the future, the sustainability question might transform itself, because a university withour OER would receive less and less students.

This conclusion means that it is vital for universities to invest in OER. In times of financial cuts by governments, methods to produce new material as OER at minimal cost are increasingly important.

9.3 An example OER product

Before we introduce our CPD method for producing sustainable OER, we describe our first product that we created using this method, as a running example. This first product is a free course on the programming language Scala. Because the material we will be discussing is targeted at the (potential) students of our university, it is at an academic level. However, our method probably does not have restrictions with respect to the level of education.

9.3.1 Scala

The design goal for the programming language Scala was to eliminate the need for different languages for different goals [Odersky et al. 2010]. For writing a script,

[12] MIT Open Courseware, `http://ocw.mit.edu`

for instance, one needs a different programming language than for writing a compiled program. For programming in an object-oriented style, one needs a different language than for programming in a functional style. For creating a special purpose (or domain-specific) language, one needs a different programming language than a general purpose language. In many cases, a special language is needed for parallel programs as well. Scala has been designed to fulfil all these purposes.

On the one hand, these features make Scala easy to learn. The language can be learned by starting with very simple scripts, without the need to write a complete program (which always involves the use of classes and objects). Using the Read-Eval-Print-Loop (REPL), a student may start by writing single lines in Scala, and one can immediately see the results.

On the other hand, these features make Scala difficult to learn, because of the diversity of language features. The object-oriented part of Scala, for instance, has enhancements to the features of Java and C#. In Scala it is possible to create an object without an associated class, Scala has the notion of a companion object and a companion class, Scala has traits, case classes, and case objects, and all these features have to be learned. The same applies for the functional programming part of Scala. On top of that, even for students who are already familiar with both an object-oriented language and a functional language, it is difficult to learn when to use the object-oriented aspects, and when to use the functional aspects. And of course, there are features for writing parallel programs, and features enabling one to develop a domain-specific language with Scala. All those aspects require not only knowledge about syntax and the API, but also knowledge about best practices, and when and how to use each feature.

Scala is an attractive language for educational purposes, because of its many features. In a course about concepts of programming languages, for example, Scala can be used to show examples of a variety of concepts. Scala could be used as a first programming language, by starting with writing scripts and explaining the concepts of variables and functions, later adding object-orientation, functional programming, and parallel and distributed programming. Scala would then become the language used in a variety of courses. The disadvantage of such an approach (besides the fact that all those courses would have to be rewritten completely) is that the main programming language of the curriculum would be a language that is not widely used at this moment, although the list of companies adopting Scala is growing, and contains successful companies like Twitter or LinkedIn.

What we did want, however, was to create the possibility for our students to get acquainted with at least several features of Scala. Therefore, we decided to create a short course on Scala, in which we assume knowledge of an object-oriented language such as Java as a prerequisite.

The course we created [Stuurman and Heeren 2012] can be read on a Wiki (in 87 pages) or downloaded as a pdf document (of 124 pages). We estimate that someone with a good understanding of an object-oriented language such as Java needs about 30 hours to study the course.

The course is divided into four Sections: 'The Basics', 'Object-orientation', 'Functional Programming' and a project. Each section has an introduction, learning goals

that direct the student, a body with text and exercises, feedback on the exercises, and a summary.

The last section of the course consists of the exercises we used at the initial meeting with the students. The exercises suggest modifications to a basic implementation of the snake game. For the course, we wrote a complete set of answers to the exercises, so whoever gets stuck can see how the exercises can be solved.

9.4 The CPD method for sustainable OER

In this section we discuss three essential ingredients for creating OER course material using the CPD method: organizing a reading group of teachers around a topic for which professional development of these teachers is considered to be required, an early meeting with interested students, and continuous feedback by students on the course material.

9.4.1 Reading group

Teachers at a university have to keep their knowledge up-to-date: updating your knowledge is part of the continuing professional development, which is a responsibility of every university teacher. This is especially true for a fast-changing domain such as Computer Science. This task can be implemented by forming a reading group around a topic of interest. Organizing such a reading group involves activities such as choosing a textbook or set of articles, preparing presentations for each other, and discussing the content in a number of sessions. To engage in such a reading group from time to time should be part of the job.

In the case of the Scala reading group, we also organized a joint programming day with the participating teachers. The idea of such an event is to work out a bigger program than the typical short examples found in textbooks, and to share some practical programming skills. This is an implementation of Kennedy's community of practice model for continuing professional development [Kennedy 2005].

Instead of keeping this newly acquired knowledge for ourselves, the method we propose here for developing OER prescribes that an additional effort is made to transform the lecture notes and the prepared exercises into a short course. In our first OER course on Scala we made this effort afterwards; for future projects we suggest to make such a transformation each time after a session. In our setting, the course consists of a collection of pages in a Wiki, which makes it easy to extend the content, to make adjustments afterwards, or to give feedback on each other's work.

Whether this transformation of lecture notes into course pages in a Wiki is extra work or not is a difficult question: by transforming sheets and private notes into public pages on a Wiki, the teacher is forced to get a deeper understanding of the subject than when preparing a presentation for fellow teachers who have read the same textbook or article. By adopting the practice of valorisation into an open course, the members of the reading group thus acquire deeper knowledge of the subject they study than would have been the case otherwise. Over a period of time, this means a transition from broad knowledge into both broad and deep knowledge.

One could state that the transformation of lecture notes into course pages does require extra work, but that the teacher benefits from this extra work, acquiring deeper knowledge.

With the CPD method, a course can be prepared almost without additional cost. Hence, the process provides a way to create sustainable OER. For the Scala OER course, we did get support from the faculty (200 hours) to create a course of the same quality as our regular courses. These extra hours were used to write learning goals, summaries, and in particular many programming exercises with feedback and a solution model for the programming assignment. In our opinion, one can create a course without these extra hours; in that case, the material will probably not contain exercises, and will be more 'raw' in appearance.

9.4.2 Meeting with students

One of the requirements for the development of an open course is the interaction with future users in the process. Such interaction should start in an early stage. Below, we explain how this was implemented in the Scala OER.

Because students of a distance learning university rarely meet each other or the teaching staff, the Computer Science faculty of the Open Universiteit organizes three meetings a year for her students. Each meeting explores a popular topic. In general, members of our staff contribute to these days, and often, experts are invited to give a guest presentation. On one such an occasion, we provided the students with an introduction to the Scala programming language (the topic studied by the reading group). After an introduction to the basic concepts of the language, an experienced Scala programmer explained the fundamentals of the Scala Lift web-framework. In the afternoon we organized a Scala workshop around the program that was written during the reading group's programming event.

For the workshop, we decided to provide an implementation of the classic snake game as a starting point, with only basic functionality. Students were asked to program a number of extensions, such as detecting that the snake bites itself, food disappearing after a configurable number of seconds, and so on. We had some idea of how to implement those extensions, but we organized the workshop without having everything worked out ourselves. This turned out to be sufficient. Afterwards, we worked on a model solution (note that this was possible because of the extra funding) and published this online, together with the OER course.

The meeting made it very clear that the selected subject attracted a considerable amount of attention. The first indication was the number of students registering for the meeting. While these meetings generally attract around 30 students, the meeting with Scala as a subject attracted more than 50 students. The second indication was the fact that most students stayed until the end of the workshop (from 10 in the morning until 5 in the afternoon), struggling with exercises in Scala. The attending students were characteristic for the students of a distance learning university: adults with a full-time job, studying in the evening hours, and many of them with a family. They not only attended this meeting on a free Saturday, but they also spent their energy on listening to lectures on a far from trivial level, and on trying to complete the exercises in Scala we challenged them to make. They were enthusiastic, and

wanted to learn more about the language. This meeting made it very clear that a course on Scala would be welcomed by a part of our student population.

9.4.3 Continuing feedback

There are several possible ways to engage students in the creation of educational material. One possibility is to ask students what they would like to learn, for instance, by organizing brainstorm sessions. Also, Wikis can be used in several ways as a collaborative medium, for instance, to enable students to prepare lecture notes in a collaborative way, to construct a library of algorithmic problems and solutions, or to collaboratively edit a textbook [Homola and Kubincova 2009].

In our case, we had the choice between trying to have students construct a course on Scala by themselves, for instance using a Wiki, or to start writing course material ourselves, and to use the Wiki to facilitate feedback by students. The first option is possible when enough students are willing to spend a lot of time in trying to learn the language using various sources, and to collaborate on a course. In our particular case, we expected that this option would not be viable. The course on Scala is not part of the curriculum, hence studying the language does not help completing the curriculum. Second, the language is relatively new, which implies that the number of people with some understanding in the topic is limited, especially when restricted to the Dutch-speaking population. At last, the Scala language is very rich and supports many features, which makes it difficult to select coherent parts for self-study.

Research on the use of Wikis in education shows that Wikis are useful for 'negotiated meaning', where a Wiki page deals with one clear subject and students can comment on the content [Bower et al. 2006]. Wikis are known to be less suitable for tasks regarding unstructured information, such as writing a new course. Such findings also point out that Wikis are useful for getting feedback on course material, but less useful for creating a new course from scratch. The same conclusion was drawn from an experiment by Cole, in which students were very reluctant to publish something on a Wiki, especially when there was no clear simple task to perform, and in which the time invested was long in comparison to how students would benefit from the contributions to the Wiki [Cole 2009].

These findings support our decision to develop the course ourselves, to use a Wiki for the content, and to ask readers to give us feedback by leaving messages on Wiki pages, for instance because an explanation was not clear to them, or because a mistake was found. The technical choice for a Wiki makes such a collaboration possible; the platform we use makes it even more attractive because all messages are from registered users. The user profiles of students that interact give us some insights in the students studying the OER.

9.5 Characteristics of the CPD method

Here, we discuss the characteristics of our method with respect to the aspects of funding, technical platform, content, and staffing.

Funding

We cannot rely on external funding, and want to offer OER for free, which rules out other types of funding. In some cases, there might be a small amount of funding from the institution itself, but the focus is on minimizing the costs.

We minimize those costs in two ways. First, we do not use existing material for our OER courses, thus minimizing the risk of losing paying customers by offering the material for free. We use new material. Second, we create OER based on new material as a side-product of Continuous Professional Development activities that are already performed by the teaching staff on a regular basis. University teachers, especially those within the dynamic domain of Computer Science, continuously have to update their knowledge. Ideally, a free course should be the valorisation of that effort: instead of keeping the acquired knowledge to themselves, university teachers could materialize their newly gained knowledge in the form of a free course.

Ideally speaking, the open courses should have the same quality as courses within the curriculum. One of the arguments for OER is that it functions as an advertisement for our courses and programs. That works best if the free course shows the same quality offered in regular courses. Another cost-related aspect is that it would be an advantage if parts of the open courses could be used in (future) regular courses. Preparing courses with the same quality as regular courses though, is only possible with extra funding. Courses that are created as a side-product of CPD probably will not contain exercises and solutions, for instance.

Relating this approach to the existing funding models, we simply try to stay outside those models, and have found ways to reduce the costs. We achieve this by creating courses as side-products of efforts we already make. In this way the additional costs of production are virtually zero. Improving the course upon student feedback will require minimal costs that can still be considered part of the CPD.

Technical

Choosing a technical platform may seem to be of minor importance, but the availability of a platform that enables the type of OER you want to establish is vital for success. In the case of the Open University of the Netherlands, such a platform is available. OpenU[13], based on Liferay Portal[14], is used by the faculty of Computer Science to present courses, research activities, staff members, and news items to the public. OpenU comes with some typical social media features, such as profiles for registered users, and the possibility to blog or to use Wikis [Schuwer, Lane et al. 2011]. This platform allows us to create OER in a flexible way, and simplifies interactions with users. The platform is accessible for everybody, and content can be viewed without registering. Furthermore, registration is free and possible for everybody. Registration enables users to publish comments to the content.

In our view, there is a third alternative to the two existing models of offering OER 'as is' or allowing users to download, adapt, and send back resources. This alternative is to allow users to give feedback and to interact with the content providers. A

[13] http://portal.ou.nl/

[14] http://www.liferay.com/products/liferay-portal/

requirement for this approach is the availability of a social platform with a Wiki. We consider this third alternative as an essential element of our CPD method. The feedback of students serves as feedback to the professional teacher. Some subjects may not have been understood thoroughly by the teacher, which may lead to comments of students. When the teacher improves the course on the Wiki, the understanding of the teacher improves.

A disadvantage of the Wiki as it is offered by OpenU, is that it is not possible to receive notifications of comments placed on a Wiki page. This means that, in practice, one shoould regularly check all Wiki pages for new comments. The possibility of notifications is, thus, highly desirable.

Content

In our situation, it is considered important that OER courses differ from the courses within the curriculum. The method we propose is intended for creating *new* courses, on *new* subjects.

Our approach is to engage students in an early stage, both to check whether the selected subject is of interest to the students and to receive early feedback. Such interaction is not only important for the development of OER, but it also serves a more general purpose. Interaction motivates students of a distance learning university to continue their study, which can be very hard for students with a full-time job, having to study in the evening hours, or for students who are confronted with all kinds of circumstances in their personal situation.

The subject of an open course should be a new development within the field, not covered in the regular curriculum. The domain of Computer Science is a domain with fast-paced changes. Even though courses are aimed at the steady theoretical base of the domain, it is desirable to pay attention to recent changes, if only to keep students interested. To develop a new, regular course is labor-intensive (and hence costly). On top of that, we can only introduce a new course when we withdraw another one from the curriculum: short open courses offer an attractive alternative for covering hot topics without giving up the more fundamental courses in a curriculum. A short, free course offers students to get acquainted with new developments.

These requirements for the content of an OER are not covered by the existing content models. Engaging students in an early stage is in line with the recommendation of the OER Ithaka report [Guthrie et al. 2008] to understand users' needs.

Staffing

The OER course is written by our own staff, but we engage students in an early stage, and ask for continuous feedback. This can be seen as a combination of the producer-consumer model and the co-producer model.

9.6 Satisfying CPD subject criteria

Our requirements concerning the subject are, as we have explained in Section 9.5, that it should persuade students to engage, that it should not (yet) fit in our regular curriculum (because we would create a regular course in that case, not an OER course), that it concerns a recent development, and, most important, that there are several teachers who want to gain knowledge on this subject. One example of such a subject is the programming language Scala.

9.6.1 How Scala meets our requirements

Scala has been around for some time now: its development started in 2001 and its first release was in 2003. However, adoption of Scala by enterprises is of more recent origin, and has grown significantly since the creators of the language received funding, which made it possible to launch Typesafe[15], a company providing commercial support, training, and services for Scala. Functional programming in general gains attention (slowly) from outside the academic domain, and Scala is one of the factors in that growing attention. So Scala can be seen as a new development, which is attractive for current and future Computer Science students.

Scala is also an attractive subject for the reading group: we teach our students to program in Java, and as teachers, we want to explore alternatives to be prepared for making a switch to another programming language. It seems too early to use Scala for that purpose, but it is certainly valuable to have knowledge of Scala for a university teacher with programming languages as one of the subjects.

Having an open course on Scala would also make it possible to use parts of it in regular courses, for instance, in a course on principles of programming languages. It would be an advantage if we could point interested students to a free course on the subject.

9.7 Evaluation and Conclusion

We first evaluate the use of the CPD method , and then the engagement of students with Scala course. At last, we draw conclusions.

9.7.1 CPD method evaluation

We evaluate the Continuous Professional Development (CPD) method for sustainable development of Open Educational Resources by answering the four questions we posed in the introduction.

The first question was to identify the characteristics of the CPD method we propose to develop an OER course, and how it differs from existing models of sustainable OER development. One characteristic is that we try to minimize the costs by creating a

[15] http://typesafe.com/company

course as a side-product of activities that are carried out anyway, as part of regular continuous professional development. This approach differs from existing funding models. We think it is important to make the knowledge that individual teachers acquire when keeping up in their field of work, available to the public. Another characteristic is that we require a social platform with a Wiki for our OER. This characteristic differs from existing technical models. A third characteristic is that we develop new material for OER, that the subject should meet several criteria (not covered by the curriculum, a recent development, interesting both for students and the staff studying the subject), and that we engage students early on. This approach differs from existing content models. A fourth characteristic is that the OER course is produced by staff members, but that students are engaged in an early stage, and that we ask for continuous feedback. This is a combination of the producer-consumer model and the co-producer model.

The second question queries whether it is possible to engage students and non-students during the development of a course. What works and what does not work, in that respect? The purpose of a meeting in an early stage is to check the attractiveness of the selected subject. It also prepares students on what is to come, and will make it more likely that they will provide active feedback. A Wiki with the possibility to add comments works very well, but in our case the disadvantage was that we regularly had to check all pages for comments, due to the lack of a notification mechanism.. In this case, a forum to discuss the course did not work. The threshold to expose yourself appears to be too high (which is supported by the fact that we also received feedback by email, with the explicit statement that giving feedback in public did not feel right). As for the cooperation of students in constructing a course in a setting without mandatory exercises, one should not set the expectations too high. The fact that students do give feedback reflects their interest, and the fact that one is able to give feedback stimulates an active attitude while studying.

The third question concerns the criteria for selecting a subject that is suitable for this type of OER development. We have described our requirements earlier. Another example of a subject meeting the criteria is the development of mobile applications for the Android platform. At the moment, we are following the same process with this subject: we are organizing meetings around this topic, and are reading material, giving talks, and programming.

The last question asks for the advantages and disadvantages of the CPD method, and the resulting OER. In our opinion, it is worthwhile to spend some of the time that each teacher devotes to continuous professional development to materialize the acquired knowledge into OER material. The question is, however, to what standard. In the case of the open Scala course, we were able to put in 200 extra hours to create a course that resembles our regular course material in structure and in quality: material that can be studied at home without any extra guidance. As a comparison, to create a regular 100-hours course from scratch we generally need more than 2000 hours: 200 hours of work for a 30-hours course is a low investment. These extra hours are not always available. When one would like to use this method to create an OER course without the funding for extra hours, the trade-in will be quality: a course created by different authors in a Wiki, to reflect what they learned about a new subject, will not be as consistent and as logical in structure as our Scala course,

and will lack exercises with feedback. We think that even the buzz it creates in social media may make it worthwhile.

9.7.2 Scala course website evaluation

Staff members started to add feedback on the first couple of pages, to lower the threshold for students to do the same. This approach worked. Students started to give feedback on subsequent pages. Sometimes the feedback had the form of a question ('I don't understand x, is it such that y?'); sometimes it had the form of a suggestion ('I miss attention for z'). In some cases, students started to help each other by answering questions. The total number of comments, however, is modest (42 at the time of writing), with only 9 students writing the comments (staff members not included). The last comment, at the moment of writing, is from one week ago, more than half a year after the release of the course. On top of these 9 students, 3 students sent comments by email.

These numbers do not imply that the open course has few readers. Nielsen concludes that in online communities, 90 percent of the users does not actively contribute anything, 9 percent contributes something once or a few times, and only 1 percent of the users actively contributes to the community [Nielsen 2006]. The same applies to Usenet [Whittaker et al. 1998]: only a few users actively contribute. Given the fact that this course is not part of a community, but is a given piece of information on which people may comment, we think that the amount of feedback we received can even be considered high. This is in accordance to the engagement that our own students showed during the meeting.

Since the course came online, we had about 15,000 page views, in 8 months. The course contains 87 pages, which means there was a mean of about 72 page views per page (but obviously, the first pages of the course were viewed more often than pages deeper in the course). For instance, the opening page of the second part about object-orientation was visited 385 times, and the opening page of the third part about functional programming was viewed 357 times. There were 414 downloads of the pdf document containing the whole course. These numbers clearly show that the course interests people, in particular when one realizes that the course is entirely in Dutch.

It was interesting to see where our users come from. Many of them followed links on our own web pages, about half of them found the course using search engines, and many users came from various places where the course has been mentioned, including LinkedIn, Twitter, Facebook, other universities, and various other sites. Also, we noticed that people have emailed each other about the course, and followed a link in the email. Obviously, this is good news for the name and credits of our university.

9.7.3 Conclusion

We have presented a new method for creating OER with as main characteristic that OER are created as a side-effect of Continuous Professional Development. We compared the method to existing methods and applied it to create an OER on the Scala

language. This course was followed by enthusiastic students who improved the course via feedback on a teacher-created Wiki. The course received considerable attention. All in all, the experience with the Scala course confirmed the usefulness of the use of the CPD method for creating OER.

Part III

Epilogue

10

Epilogue and future work

10.1 Summary

The subject of this thesis is design for change. We discerned two strategies for design for change: applying variation points and enhancing changeability. These strategies do not rule each other out: they may be (and often are) combined. We addressed both technical and educational aspects of these strategies.

10.1.1 Variation points

In Chapter 2, we described new constructs offered by mobile platforms, such as Android, that make it possible for mobile apps to react to changes in the environment or to changes of the device itself. An example is communication through intents, which makes it possible for mobile apps to communicate with other apps without knowing of their existence. We showed that these new constructs need new modeling techniques. This fact has implications for Computer Science curricula. Intents, when used undirected, are a form of variation points: they decouple the service an app requires from the identity of the app that may deliver the service.

Chapter 2 relates to this thesis because mobile platforms are designed with pre-built variation in mind. New constructs in mobile platforms make it easier to build applications with pre-built variations, but it is not straightforward which modeling techniques might be used to design such applications. The chapter also relates to the educational aspect.

In Chapter 3, we described the problem of using design patterns to teach students to design flexible object-oriented programs: it is difficult to see the value of design patterns in isolated, small exercises. We found a solution by having students work in pairs, extending an existent program. Design patterns are a form of best practice, coupling general solutions to general problems. Design patterns are, in general,

a combination of decoupling and abstraction in a specific setting. Design patterns decouple variations from the element that uses these variations, by allowing the using element to use a variation without being aware which exact variation it is actually using. As such, they form variation points: it is easy to add variations.

Chapter 3 relates to this thesis because it discusses the educational aspect of teaching design patterns, while design patterns help in creating variation points.

Chapter 4 describes two different mechanisms to apply a change with respect to traffic rules to a distributed control system for unmanned vehicles at run-time: by replacing a process with a different process, or by injecting new versions of Java class files and forcing the processes to use these new classes. In the first case, the processes are the variation points; in the second case, Java classes form the variation points. Processes apply uniform traffic rules in case of a version mismatch. Both techniques have advantages and disadvantages. An advantage of the class-based change mechanism is that the subscription-based architectural style can be maintained: there is no need for one process that has a direct connection with all other processes. Processes thus are all decoupled from each other.

The link with this thesis is the fact that it is about dynamic software updating. Changes that are necessary from time to time, without stopping the system, are new traffic rules for the vehicles. We showed the consequences for the architecture of the choice of variation points: processes or Java classes.

In Chapter 5, we described a system for intelligent feedback for students working on exercises, in which several strategies for design for change are combined. Web services form a variation point, decoupling user interfaces from computing feedback. Strategies for solving a specific kind of exercises and rules within a specific domain are declared using an EDSL, separated from the remainder of the source code: another example of variation points, this time created using abstraction. The source code (as well as the EDSL) is written in the functional programming language Haskell, and prevents side-effects by refraining from a global state. The same applies to the web services that are created along the principles of the REST architecture, which demands stateless communication.

The framework described here is an example of the strategy to separate the code that will probably have to be changed often, from the rest of the code. It is also an example of the strategy to reduce side-effects, and to apply loose coupling in the form of web services.

Chapter 6 describes guidelines for students to create programs in JavaScript that adhere to the design principles of the Software Engineering Body of Knowledge. These guidelines have the form of steps in refactoring, and checking how future changes would effect the code is an integral part of these steps. We showed that designing with changeability as a goal has a positive influence on the quality of the code. The resulting code should show a high degree of abstraction and a high degree of decoupling. In the example problem, we declare changes outside the sourcecode, in HTML.

Chapter 6 addresses the educational aspect of design for change. It shows that design for change may help in deriving elegant, flexible code. In particular, the strategy to declare changes outside the code gets attention.

10.1.2 Enhancing changeability

In Chapter 7, two software architectures for a railroad control system are described, and analyzed with respect to several quality requirements such as changeability. We show that the choice for a software architectural style influences those qualities, and also that the choice of the modeling technique used to model the problem influences the software architecture. This means that, in the end, the choice for a certain technique to model a problem influences quality aspects of the resulting system.

The relevance to this thesis is that architectural styles differ with respect to changeability, and that the choice for a modeling technique to model the desired behavior of the system is intertwined with the choice for an architectural style. The choice for such a modeling technique therefore, in the end, influences non-functional properties such as changeability.

Chapter 8 describes the radio-broadcast architectural style and the Java implementation that we developed. This implementation is described in a formal abstract language, which makes it possible to reason about certain quality properties of a system designed in this architectural style. The architectural style uses a form of communication that decouples the communicating processes from each other.

The architectural style that is chosen for this framework, together with the formal language that can be used to describe the resulting system, allows one to reason about desired properties of the system. The architectural style is the Radio Broadcast Paradigm. The possibility to apply changes at run-time in this style is discussed in Chapter 4.

In Chapter 9, we showed a method to create short courses on subjects that do not (yet) fit within a current curriculum, as a side-effect of the professional development of teachers. This allows one to create courses on, for instance, Scala, a programming language that eases the use of a functional programming style for developers with an object-oriented background. A functional programming style, in which one refrains from maintaining a global state, enhances changeability, because one avoids the rippling effect of introduced bugs. The focus in this chapter is on the process of creating short courses; the link to this thesis is the fact that a short course on Scala contributes to the ease with which developers may enhance changeability in their applications. Our method to create short courses also illustrates how to enhance the changeability of a curriculum.

10.2 Observations

The problem to find a method to design for change is not solved once and for all. On the contrary, design for change is a subject that will never be ‘solved’. The fact that the problem will never be solved has several causes. Design is inherently hard in the first place, because it involves conquering the problem of complexity. Design with ease of change as one of the goals adds complexity. Another cause is formed by ongoing developments in programming languages and programming platforms, which show an increase in abstraction. Such abstract concepts in programming languages demand new approaches to use those concepts. A third cause is the fact that

systems get more complex, and the more complex systems are, the harder design for change is (the harder design is in general).

When looking back to our contributions in this field, with respect to variation points, to enhancing changeability, and to educational aspects, we think that the following observations can be made.

10.2.1 Modeling diagrams

New technologies bring new concepts. Programming languages (Scala is an example) and programming platforms (Android is an example) tend, over time, to offer more abstract concepts to the programmer. These abstract concepts may help in conquering complexity and in achieving a higher degree of changeability. On the other hand, systems tend to grow more complex: these extra facilities of languages are needed to be able to build more complex systems.

Invariably, modeling techniques lag behind the development of technologies. This phenomenon can be seen, for instance, in the Android platform, where the concept of an intent lacks a modeling construct (see Chapter 2). The same applies to the technology of JavaBeans, years ago [Stuurman 1999]. The lack of modeling techniques may be observed in many places. For instance, there is no obvious way to model C# delegates in UML, the modeling language that is universally used to model object-oriented programs [Henderson-Sellers 2005]. The same applies to communication through events using delegates. Also in Java, it is unclear how to model implicit associations between classes (for instance, in the case of communication through events) in a class diagram. In Scala, one may use objects (singletons), classes and functions in one program; it is unclear how to model those three constructs in UML. The same applies to JavaScript, where objects, constructors and (other) functions may be combined in a program. It is also unclear how to model JavaScript modules.

Of course, this is not something to be surprised of: new techniques first have to establish themselves, and new modeling constructs or notations and languages should be proposed, discussed, used and evaluated before there can be a general agreement on how to model those techniques.

10.2.2 Implicit design

We saw, in Chapter 7, that the choice of a modeling technique to analyze a problem (in this case event-based versus state-based) leads to two different architectural styles, with different influences on quality properties of the resulting system. This fact is, in particular, important for automated programming, where implicit design is, in general, part of every step in the process. A very simple example of implicit design is the Rational Unified Process [Kruchten 2004], in which analysis transforms into design, using class diagrams, which will lead to an object-oriented design as opposed to, for instance, a functional design.

Often, decisions in the development of systems with implicit design are not based on the implications for the design, but on other considerations: the availability of tools to reason about the problem for different modeling techniques, for instance, or the knowledge and skills of analysts and designers. However, one should take

the implications for the resulting design into account when making decisions about the tools and techniques to model the problem. In the end, these decisions have implications on, for instance, the changeability of the resulting system.

To take these implications into account, they should be known, and this is not always the case. Summarizing, one may argue that implicit design should be made explicit where possible.

10.2.3 Facilities of languages and isolation of change

We have seen two examples in which change was isolated. In a framework for intelligent feedback for exercises, the domain and strategies to solve exercises were declared using an EDSL (see Chapter 5). In an example in JavaScript, changes in forms were isolated in HTML, which made it possible to use the same JavaScript code for every form imaginable (see Chapter 6).

To isolate change, one may:

- use a particular place in the code where changes have to be applied (factories are an example),
- use an EDSL to declare the changes (see Chapter 5),
- use a Domain Specific Language (DSL) to declare the changes, and write a parser to read the changes,
- use an existing language (see Chapter 6) or create a text format (such as in a configuration file) or a file format (such as in a spreadsheet program). In these cases too, a parser must be written (in the example of JavaScript and HTML, this task is performed by the browser).

Embedded Domain Specific Languages (in which the declared changes are compiled together with the remainder of the code) and Domain Specific Languages (that will be parsed by the program) have different advantages and disadvantages. Using an EDSL, one may use the facilities offered by the underlying programming language, for instance. Interpretation versus compilation has consequences for performance. On the other hand, using a DSL, it is possible to allow users to declare changes. Advantages and disadvantages of this approach have been discussed for the framework for exercise assistants, on the basis of a number of case studies [Heeren and Jeuring 2010].

One of the examples of the strategy to enhance changeability is automated programming, in which bugs are prevented by generating code using a code generator that has proven to be correct [Toetenel et al. 1996]. In fact, this strategy may be seen as an example of the strategy to isolate change, in which everything is supposed to be changeable. This raises the question of where to draw the line: to which extent do we isolate changes?

One may view programming in a programming language as a form of automated programming: the compiler produces the machine code, while we have declared what we want the resulting program to achieve using a higher-level programming

language. This fact may help in deciding what to include in the 'permanent' code, and what to declare as changes. What is declared as changes, is declared using another language (although such a language may be based on the programming language in the case of an EDSL). Applying changes in this language should be easier than applying the same changes in the code.

In the case of a DSL, embedded or not, the choice is obvious: the language is geared to a specific domain, and that is what the language should be used for. For everything outside this domain, the programming language should be used.

In other cases, the choice partly depends on volume: programming languages tend to be prepared for a high volume of code. For instance, name space, packages or modules, information hiding and abstraction are almost always part of a programming language. In some cases, languages to declare changes outside the code lack such facilities. When the volume of lines is low and will stay low, this is not a problem, but when the volume is high, the same kind of facilities are required of the declaration language.

10.2.4 Best practice

Best practice refers to experience-based strategies for problem solving, in which the solution is not guaranteed to be optimal, but 'good enough'. We have seen that modeling techniques lag behind with respect to new facilities in languages. On a deeper level, there is another problem with new technologies: it takes a long time before there is a general agreement on how to use new concepts, and on what makes a design that uses these concepts a good design or a bad design. Even when such an agreement has been established, it may take a long time before there are best practices available on how to solve problems with these new concepts. An example is the question how to combine object-oriented and functional aspects of the language Scala (see Chapter 9).

Design patterns are a form of best practice (see Chapter 3). They were formulated a long time after the appearance of the first object-oriented languages: it takes time before there is a general agreement on how to use language features in a 'good' way. However, students still struggle to recognize the general problems in a given problem: there is a need for guidelines that guide one in how to use design patterns when trying to design a system: there is still no common ground on that question.

One may rephrase one of the problems of implicit design into terms of best practice as well: what is needed, is a set of best practices that guide one to the right modeling technique for a problem, given the (quality) requirements of the desired solution.

10.2.5 Global state

One of the strategies to enhance changeability is, as we have seen, to refrain (as much as possible) from a global state, to prevent a ripple effect of bugs. On the other hand, we saw that an architectural style with a global state did have advantages (see Chapter 4). This raises a question: should one always avoid a global state, or is a global state 'harmless' in some situations?

The answer may be associated with the restrictions for subscription-based communication:

- The order in which the data are sent is not necessarily the same order at which they are received.
- There is no way to ensure that data are not lost, other then sending some precious data twice or more times.

The typical use of subscription-based communication is where the same kind of data is sent over and over again, every time slightly different [Boasson 1998]. In such a case, the loss of one message is no disaster: the receiving processes temporarily use information slightly older than it should be.

The same applies to a global state: when components update the global state regularly, each time modifying some of the variables in the global state by a fraction, one may safely use a global state. Steels describes how he uses a global state in this manner for his Lego robots: different sensors add or subtract from variables in the global state, and these variables direct the actions of the robot [Steels 2003].

In other cases, where changes of the global state are more far-reaching (a simple example is a variable that is either `true` or `false`), or when changes are not applied over and over again, a global state has the disadvantage that a bug in one of the components may influence components that read the global state, and the rippling effects make the software hard to debug.

10.2.6 Education

We have presented various contributions on the wide subject of design for change. We showed that there are two strategies to design for change, and that these strategies may be used simultaneously: apply variation points and enhance changeability. The 'tools' for both strategies are decoupling and abstraction. These strategies and tools, one may safely assume, are the invariants in software design for change. The fact that these strategies and tools are known, does not mean that it is clear how to use those strategies and tools: each situation is different, and the study of design for change consists for a great part in finding best practices with respect to, for instance, where to apply variation points, and how.

Best practice is closely related to education. In Chapter 6, for instance, we described how we supply students with guidelines telling them how to refactor a given program to a program that complies to the software engineering design principles. Another example is that we use design patterns as a means to teach object-oriented software design (see Chapter 3).

Making explicit how to solve problems, and supply students with procedural information in the form of guidelines and best practices, is one technique to teach students how to design to change; working in a setting similar to an architect's studio, with the teacher as the master and the students as apprentices, is another technique (see Chapter 3). Where procedural guidance can be made explicit, we should do so, in particular in distance education, but where this is not possible, the master-apprentice approach allows us to convey implicit knowledge.

In practice, software design should be a synonym for software design for change. Almost any software design should be optimized for change. An exception might be a product that is thrown away, and is completely redesigned from scratch after the first try, but in such a case, the first try might better be called a prototype that functions as an input for the design of the first product. Education of software design is therefore a synonym for education of software design for change.

10.3 Future work

Each of the observations we made, may lead to future work. The lack of suitable modeling techniques with respect to new features of programming languages or programming environments might lead to the research of modeling techniques. The observation that implicit design should be made explicit, might lead to research into the implications of the choice of different techniques for analyzing problems.

With respect to education, we are interested in how heuristics may help students learn to design software at all levels (programming, design, software architecture). We think that we should make an effort to develop heuristics (or making implicit heuristics explicit) for different forms of design, to effectively teach software design. We suspect that explicit guidelines might help students to learn to design. Our future work will be to explore explicit guidelines: in which areas are we able to formulate guidelines? Do guidelines indeed help students to grasp design?

Another benefit of explicit strategies to design a solution for a problem, is that one of the conditions for automatic semantic feedback while solving exercises is met. We would like te explore the possibilities of providing students with rich feedback while designing an object-oriented solution for a problem using UML.

After all, now and in the future, the only way to ensure that software systems become more optimized with respect to future changes, is to equip students with the right knowledge and experience, because they are the software engineers of the future.

Samenvatting

Stel, dat ICT-projecten als bij toverslag netjes binnen het gestelde budget zouden opleveren wat de opdrachtgever wenst. Zelfs in deze utopische situatie zouden ICT-projecten nog problemen blijven opleveren, laat ik in mijn proefschrift zien.

De wereld verandert namelijk voortdurend. Als gevolg daarvan moeten er ook steeds veranderingen worden aangebracht in software. U kunt daarbij denken aan nieuwe wetten en regels, aan reorganisaties, maar ook bijvoorbeeld aan nieuwe hardware: mobiele apparaten met hogere resoluties. Zowel de veranderende maatschappij als technologische ontwikkelingen brengen nieuwe wensen voor software met zich mee.

Software kan worden aangepast aan nieuwe wensen, maar wanneer er geen extra maatregelen zijn genomen, wordt het, elke keer dat er een verandering is aangebracht, moeilijker om een volgende verandering aan te brengen. 'Moeilijker' betekent dat het steeds kostbaarder wordt om veranderingen aan te brengen, en ook dat het risico dat er bij zo'n verandering fouten worden gemaakt steeds groter wordt. Dit fenomeen wordt ook wel 'software veroudering' genoemd.

Wanneer software bij het ontwerpen wordt geoptimaliseerd voor toekomstige veranderingen, wordt dit probleem van verouderende software zo veel mogelijk vermeden. Software tijdens het ontwerpen optimaliseren voor toekomstige veranderingen is het onderwerp van dit proefschrift. Daarbij gebruiken we 'ontwerpen' in de ruime zin van het woord: zowel software architectuur (het ontwerpen van systemen op hoog abstractieniveau), software design (het ontwerpen van software op detailniveau) als programmeren valt er onder.

Wanneer men rekening wil houden, bij het ontwerp van software, met toekomstige veranderingen, zijn er twee mogelijkheden: veranderingen zijn van te voren te voorspellen (althans, de plaats waar in de software toekomstige veranderingen moeten worden aangebracht is te voorspellen), of er valt niets te voorspellen over veranderingen.

In het eerste geval kan er gebruik gemaakt worden van zogenaamde variation points: plekken in de software die zijn voorbereid op toekomstige veranderingen. In het tweede geval kan alleen de globale 'veranderbaarheid', changeability, van de software zo groot mogelijk worden gemaakt.

De changeability kan altijd worden verhoogd, terwijl variation points alleen kunnen worden ingebouwd wanneer veranderingen voorspeld kunnen worden.

In mijn proefschrift kijk ik naast de technische aspecten van deze vormen van ontwerpen met het oog op toekomstige veranderingen ook naar de educatieve aspecten: hoe leren we studenten software zo te ontwerpen dat het resultaat zo flexibel mogelijk is met betrekking tot veranderingen?

Variation points

Software voor mobiele apparaten moet functioneren in een sterk veranderende omgeving: het apparaat kan een kwartslag draaien zodat het scherm hoger dan breed is of andersom, de verbinding kan wegvallen, het geheugen kan plotseling ontoereikend zijn, er kan een telefoontje binnenkomen, enzovoort. Software voor mobiele apparaten moet dus rekening houden met van te voren *bekende* veranderingen.

De twee belangrijkste 'gereedschappen' die de software ontwikkelaar heeft voor variation points zijn losse koppeling en abstractie. Deze gereedschappen zijn ook gebruikt in ontwikkelomgevingen voor software voor mobiele apparaten, om constructies te maken die programmeurs kunnen gebruikken om software te bouwen die op veranderingen kan reageren. In hoofdstuk 2 bespreken we hoe software voor mobiele apparaten in elkaar zit, en hoe we het ontwerpen van mobiele software kunnen opnemen in het universitaire curriculum.

Variation points kunnen worden aangebracht door losse koppelingen aan te brengen (waarbij abstractie nodig is). Het punt is dat je niet zomaar overal losse koppelingen aan kunt brengen: uiteindelijk zijn er vaste koppelingen nodig om de verschillende onderdelen van de software met elkaar te kunnen laten werken. Design patterns zijn een vorm van expliciet gemaakte ervaringskennis over in welke situatie je welke losse koppelingen kunt gebruiken. In een onderwijssituatie is het lastige van design patterns dat je ze pas op waarde leert schatten in een groot project: in kleine oefenprobleempjes bieden ze weinig extra waarde. In hoofdstuk 3 bespreken we hoe studenten aan een universiteit met afstandsonderwijs toch kunnen leren met design patterns te werken in een behoorlijk groot project.

In hoofdstuk 4 laten we een software architectuur zien voor een systeem met onbemande voertuigen. Elk voertuig heeft een eigen processor, en beschikt over de verkeersregels. Een van de voor de hand liggende toekomstige veranderingen is het veranderen van de verkeersregels (bijvoorbeeld als gevolg van schaalvergroting, of van de introductie van nieuwe soorten voertuigen). We bespreken twee manieren om nieuwe verkeersregels in het systeem aan te brengen, waarbij de ene manier voldoet aan de gegeven software architectuur, terwijl de architectuur voor de andere manier moet worden aangepast. We bespreken voor- en nadelen van beide manieren.

Voor studenten is het bij allerlei oefeningen (bijvoorbeeld wiskundige vergelijkingen oplossen, logische formules in een normaalvorm zetten, of kleine program-

meeropdrachten) handig om te kunnen oefenen in een situatie waarin er directe feedback komt. Aan de Open Universiteit hebben we, samen met de Universiteit Utrecht, een tool ontwikkeld dat het mogelijk maakt om studenten semantisch rijke feedback te geven bij het oplossen van dit soort problemen. Toekomstige veranderingen zitten hem in het domein, in de regels binnen dat domein, en in oplossingsstrategieën. Zoals we in hoofdstuk 5 bespreken, zijn die aspecten in de software geïsoleerd, en worden gespecificeerd in een domein-specifieke taal. Dat maakt toekomstige uitbreidingen gemakkelijker. Een ander aspect is dat de feedback wordt aangeboden in de vorm van webservices, waar allerlei front-ends gebruik van kunnen maken. Ook dat is een vorm van losse koppeling die het mogelijk maakt heel gemakkelijk nieuwe front-ends in gebruik te nemen.

In hoofdstuk 6 laten we zien hoe een verzameling expliciete richtlijnen voor refactoren (het ombouwen van code om deze beter onderhoudbaar te maken met het oog op toekomstige veranderingen) inderdaad leidt tot code waarin veel gemakkelijker veranderingen aangebracht kunnen worden. Een van de onderdelen van de richtlijn is om voor de hand liggende toekomstige veranderingen te bedenken, en na te gaan wat er moet gebeuren om die te implementeren. Het voorbeeld dat we hanteren laat zien hoe je wat verandert buiten de code kunt houden. Wat verandert wordt dan elders gespecificeerd, ingelezen, en daarmee kan het systeem een hele verzameling veranderingen aan zonder dat de code veranderd hoeft te worden.

Changeability vergroten

Een probleem kan op verschillende manieren gespecificeerd worden. In hoofdstuk 7 gebruiken we twee verschillende specificaties van het probleem van een controller voor een trein. We laten zien dat de ene specificatie leidt tot een software architectuur waarin data centraal staan, terwijl de andere specificatie leidt tot een software architectuur waarin functies centraal staan. We laten zien dat die twee software architecturen verschillende eigenschappen hebben op het gebied van timing, scalability, fault-tolerance en extendibility. De keuze voor een specificatiemethode voor het probleem is dus uiteindelijk van invloed op eigenschappen van het systeem, waaronder de mogelijkheden om het systeem uit te breiden.

Een manier om een systeem gemakkelijker uitbreidbaar te maken is om van te voren te kunnen bewijzen dat het resultaat bepaalde eigenschappen heeft (en zo te voorkomen dat pas na de bouw blijkt dat de gewenste eigenschappen niet aanwezig zijn). In hoofdstuk 8 beschrijven we een framework voor systemen die opgezet worden volgens het radio-broadcast paradigma, met bijbehorende implementatie. Omdat de implementatie bekend is, zijn we in staat om randvoorwaarden te stellen aan de infrastructuur van – bijvoorbeeld – een treincontroller, om te garanderen dat er – bijvoorbeeld – geen botsingen tussen treinen kunnen voorkomen. Zo'n framework maakt het gemakkelijker om in de toekomst veranderingen aan te brengen omdat allerlei belangrijke eigenschappen van te voren kunnen worden gecontroleerd.

Functionele talen hebben eigenschappen (zoals het zich zo veel mogelijk onthouden van state) die het gemakkelijker maken om een systeem uit te breiden. Als een nieuwe functie getest is, en juist bevonden, kan de introductie van zo'n nieuwe functie geen ongewenste neveneffecten elders tot gevolg hebben. De meeste pro-

grammeurs zijn opgeleid met vooral object-georiënteerde talen. De programmeertaal Scala heeft zowel functionele als object-georiënteerde aspecten, en is daardoor ideaal om programmeurs die objectgeoriënteerd kunnen programmeren om te scholen naar functioneel programmeren. In hoofdstuk 9 beschrijven we hoe we een korte cursus Scala hebben gecreëerd door vooral gebruik te maken van ons eigen leertraject daarin, in het kader van Continuous professional development. Het bestaan van die korte cursus maakte ons eigen curriculum ook gemakkelijker veranderbaar: inmiddels zijn stukken uit deze cursus in gebruik bij andere cursussen.

Dankwoord

De achtergrond van mijn promotie is – in het kort – dat ik twee keer gestart ben met een promotietraject, en dat traject beide keren heb moeten stoppen. De eerste keer moest ik mijn promotietraject staken vanwege een combinatie van het wegvallen van het onderzoeksproject, het uitvallen van de promotor, en persoonlijke omstandigheden; de tweede keer bleek dat de gevolgen van het motorongeluk dat me in 2005 overkwam toch verder strekten dan ik had gehoopt. Op dat moment heb ik me er bij neergelegd om nooit te promoveren; dat voelde als een nederlaag. Met deze achtergrond in het hoofd is het misschien te begrijpen dat ik vrijwel iedereen in mijn omgeving dankbaar ben: op de een of andere manier heeft iedereen er wel aan bijgedragen dat ik verder ben gekomen, dat ik het uiteindelijk toch heb aangedurfd om de draad op te pakken (in de allereerste plaats natuurlijk mijn levensgezel Ernst Anepool). Wanneer ik hier opnoem wie ik dankbaar ben met betrekking tot deze promotie laat ik dan ook onvermijdelijk namen weg die ik graag zou noemen. Daarom noem ik slechts enkele namen.

In de eerste plaats dank ik Marko van Eekelen, mijn promotor. Toen ik de reorganisatieplannen van onze universiteit doornam besefte ik dat niet-gepromoveerde docenten een aantal voor mij onoverkomelijke nadelen te wachten stond: er zou een demotie plaatsvinden van universitair docent naar docent, ze zouden geen tijd meer krijgen voor onderzoek, ze zouden een groot risico lopen op ontslag bij een volgende reorganisatie, en doceren in de master zou voorbehouden zijn aan docent-onderzoekers. Er zat – zo dacht ik – niets anders op dan voor een derde maal te starten met een promotietraject. Tot mijn grote verrassing meldde Marko van Eekelen dat ik voldoende had gepubliceerd om op basis van artikelen te promoveren. In 2014 heb ik bovendien nog twee extra artikelen gepubliceerd. In het traject daarna heeft hij me steeds op het juiste niveau bijgestaan: eerst op grote lijnen, en tenslotte, bij de voorbereiding van het boekje, met veel detail. Hij is voor mij de perfecte promotor.

Mijn copromotor Harrie Passier ben ik vanwege een groot aantal redenen dank-

baar. In de eerste plaats mocht ik als paranimf zijn eigen promotie meemaken, waardoor ik een goed beeld heb gekregen van wat het betekent om een boek te produceren en op het podium te staan. In de lastige tijd waarin ik de rode draad probeerde te vinden in mijn artikelen, door ze te karakteriseren op verschillende manieren en te proberen die karakteristieken in spreadsheets tot een mooie matrix om te vormen, bracht hij me op het spoor van 'the Pyramid Principle' [Minto 2009] om structuur aan te brengen in teksten, en met de structuur in de introductie ontstond ook de juiste matrix om de artikelen in onder te brengen. Hij heeft vanaf het begin veel tijd en energie gestoken in het meelezen en becommentariëren van mijn teksten. Ik ben hem ook dankbaar voor het feit dat hij iemand is van het overleg, van Skypen, van mensen bij elkaar brengen, en dat hij daarmee heeft voorkomen dat ik als een kluizenaar aan mijn proefschrift heb gewerkt. Tenslotte is hij de trekker van de onderzoeksgroep rond Vakdidactiek, waar ik na deze promotie mee verder wil.

Ook mijn andere copromotor, Bastiaan Heeren, en mijn voormalige decaan Lex Bijlsma ben ik dankbaar voor de aandacht en de zorgvuldigheid waarmee ze mijn teksten van commentaar hebben voorzien. Dankzij de opmerkingen van Erik Barendsen heb ik een aantal artikelen aangevuld, en zijn commentaar heeft er voor gezorgd dat ze een stuk beter zijn geworden. Stef Joosten dank ik dat hij mijn proefschrift mee heeft genomen op vakantie om zijn goedkeuring op tijd te kunnen geven. Serge Demeijer vroeg tijdens een visitatie, als lid van de visitatiecommissie, op zeer plezierige wijze of het niet lastig voor me was om niet gepromoveerd te zijn tussen gepromoveerde collega's. Ik ben hem dankbaar dat hij mijn proefschrift heeft willen lezen, en vooral ook voor de complimenten die hij bij zijn goedkeuring gaf. Dat was de eerste keer dat ik me echt trots voelde op wat ik heb gedaan. Ook Ruurd Kuiper wil ik danken voor de energie die hij heeft gestoken in het lezen van mijn teksten en voor de vragen die hij me daar over heeft gesteld, en voor zijn advies om vooral te genieten van de promotie en het niet alleen te zien als de weg naar een titel.

Tenslotte wil ik graag de medewerkers van de afdeling Hersenletsel van de revalidatiekliniek Adelante in Hoensbroek bedanken. Dankzij hen heb ik geleerd langzamer te werken, zoveel mogelijk gebruik te maken van externe geheugens en systemen om mijn werk te ordenen, en dankzij die hulp heb ik mijn werk kunnen volhouden en voortzetten. Met deze promotie hebben mijn hersenen, zo heb ik het gevoel, zich helemaal gerevancheerd.

Curriculum Vitae

Sylvia Stuurman

Work

2011	Elected as Teacher of the Year of the Open University of the Netherlands
2002 - now	Assistent professor of Computer Science, Open University of the Netherlands
2000 - 2002	Scientific officer, Devote, Bunnik
1997 - 2002	Researcher in Computer Science, Delft University of Technology
1995 - 1997	Research assistent, Delft University of Technology
1991 - 1995	System administrator/programmer, Delft University of Technology
1990 - 1991	Tutor, Open University of the Netherlands
1982 - 1991	Independent entrepreneur (retail)
1981 - 1983	Editor and corrector, Publishing house Het Spectrum
1979 - 1982	Teacher in Biology, secondary schools

Opleiding

1996	Master of Science Computer Science, Open University of the Netherlands
1982	Diploma for entrepeneurs
1979	Qualification to teach Biology in secondary schools ('Eerstegraads bevoegdheid'), University of Groningen
1973	Gymnasium beta, Stedelijk Lyceum Zutphen

Publications

The Design of Mobile Apps: What and How to Teach?, Sylvia Stuurman, Bernard E. van Gastel, Harrie J.M. Passier (2014) In: *Proceedings of the fourth Computer Science Education Research Conference 2014*, pages 93 – 100, ACM Digital Library [Stuurman, Passier et al. 2014]

Beautiful Javascript: How to guide students to create good and elegant code, Harrie J.M. Passier, Sylvia Stuurman, Harold Pootjes (2014) In: *Proceedings of the fourth Computer Science Education Research Conference 2014*, pages 65 – 76, ACM Digital Library [Passier, Stuurman et al. 2014]

A New Method for Sustainable Development of Open Educational Resources, Sylvia Stuurman, Marko C.J.D. van Eekelen, Bastiaan J. Heeren (2012) In: *Proceedings of the second Computer Science Education Research Conference*, pages 57–66, ACM Digital Library [Stuurman, van Eekelen et al. 2012]

Feedback Services for Exercise Assistants, Alex Gerdes, Bastiaan J. Heeren, Johan Jeuring, Sylvia Stuurman (2008) In: *Proceedings of the seventh European Conference on e-Learning*, pages 402–410, Academic Conferences Limited [Gerdes et al. 2008]

Using IDEAS in Teaching Logic, Lessons Learned, Josje S. Lodder, Harrie J.M. Passier, Sylvia Stuurman (2008) In: *Proceedings of the first International Conference on Computer Science and Software Engineering*, volume 5, pages 553–557, IEEE Computer Society Press [Lodder, Passier et al. 2008]

A Generic Framework for Developing Exercise Assistants, Johan Jeuring, Harrie J.M. Passier, Sylvia Stuurman (2007) In: *Proceedings of the eighth International Conference on Information Technology Based Higher Education and Training, ITHET*, IEEE Computer Society Press [Jeuring et al. 2007]

Turning an Interactive Tool implemented in Haskell into a Web Application – an Experience Report, Sylvia Stuurman, Johan Jeuring (2007) *UU-CS-2007-008, Universiteit Utrecht* [Stuurman and Jeuring 2007]

Experiences with Teaching Design Patterns, Sylvia Stuurman, Gert Florijn (2004) In: *ACM SIGCSE Bulletin, volume 36, number 3, pages 151–155*, and in the Proceedings of the ninth annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE), pages 151-155, ACM Digital Library [Stuurman and Florijn 2004]

Software Development and Verification of Dynamic Real-time Distributed Systems based on the Radio Broadcast Paradigm, Jan van Katwijk, Ruud De Rooij, Sylvia Stuurman, Hans J. Toetenel (2001) In: *Parallel and Distributed Computing Practices*, pages 105–126, Nova Science Publishers [van Katwijk, de Rooij et al. 2001]

Software Architecture and JavaBeans, Sylvia Stuurman (1999) In: *Proceedings of the first Working IFIP Conference on Software Architecture (WICSA1)*, pages 183–199, IEEE Computer Society Press [Stuurman 1999]

On-line Change Mechanisms: The software architectural level, Sylvia Stuurman, Jan van Katwijk (1998) *Proceedings of the sixth ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 80–86, ACM Digital Library [Stuurman and van Katwijk 1998]

Evaluation of Software Architectures for a Control System: A case study, Sylvia Stuurman, Jan van Katwijk (1997) In: *Coordination Languages and Models, Proceedings of the second International Conference COORDINATION* (Lecture Notes in Computer Science 1282), pages 157–171, Springer [Stuurman and van Katwijk 1997]

Evolving Software Architectures: A position paper, Sylvia Stuurman (1997) *Living with Inconsistency Workshop in conjunction with the International Conference on Software Engineering (ICSE)* [Stuurman 1997]

Modeling and Analysis of Complex Computer Systems-the MTCCS approach Hans J. Toetenel, Ronald F. Lutje Spelberg, Sylvia Stuurman, Jan van Katwijk (1996) In: *Proceedings of the second IEEE International Conference on Engineering of Complex Computer Systems*, pages 423–430, IEEE Computer Society Press [Toetenel et al. 1996]

MTCCS, Ronald F. Lutje Spelberg, Sylvia Stuurman, Hans J. Toetenel (1996) *Technical report, Delft University of Technology, Faculty of Technical Mathematics and Informatics* [Lutje Spelberg et al. 1996]

References

Affenzeller, Michael, Stefan Wagner, Stephan Winkler and Andreas Beham (2009). *Genetic Algorithms and Genetic Programming: Modern concepts and practical applications*. CRC Press (cited on page 6).

Allen, Julia. H., Sean J. Barnum, Robert J. Ellison, Gary McGraw and Nancy R. Mead (2009). *Software Security Engineering*. Addison-Wesley Professional (cited on pages 12, 96).

Allen, Robert and David Garlan (1994). 'Beyond Definition/Use: Architectural Interconnection'. In: *ACM SIGPLAN Notices* 29.8, pp. 35–45 (cited on page 61).

Alpert, Sherman R., Mark K. Singley and Peter G. Fairweather (1999). 'Deploying Intelligent Tutors on the Web: An Architecture and an Example'. In: *International Journal of Artificial Intelligence in Education* 10.2, pp. 183–197 (cited on page 79).

Altayeb, Badreldin and Kostadin Damevski (2013). 'Utilizing and Enhancing Software Modeling Environments to Teach Mobile Application Design'. In: *Journal of Computing Sciences in Colleges* 28.6, pp. 57–64 (cited on pages 32, 40).

Alur, Rajeev and David Dill (1994). 'A Theory of Timed Automata'. In: *Theoretical Computer Science* 126.2, pp. 183–235 (cited on page 148).

Ammerlaan, Marcel, Ronald Lutje-Spelberg and Hans J. Toetenel (1998). 'XTG – An Engineering Approach to Modelling and Analysis of Real-Time Systems'. In: *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*. Berlin, Germany: IEEE Computer Society, pp. 88–97 (cited on page 148).

Anderson, John R. (1993). *Rules of the Mind*. Routledge (cited on page 77).

d'Antoni, Susan (2008). *Open Educational Resources: The Way Forward*. International Council for Open and Distance Education. URL: `published%20online%20at%20%5Cutel%7Bhttp://openaccess.uoc.edu/webapps/o2/bitstream/10609/7163/1/Antoni_OERTheWayForward_2008_eng.pdf%7D` (cited on page 163).

Astrachan, Owen, Garrett Mitchener, Geoffrey Berry and Landon Cox (1998). 'Design Patterns: An Essential Component of CS Curricula'. In: *ACM SIGCSE Bulletin* 30.1, pp. 153–160 (cited on page 51).

Backhouse, Roland, Patrik Jansson, Johan Jeuring and Lambert Meertens (1999). 'Generic Programming'. In: *Third International School on Advanced Functional Programming (AFP),*

Revised Lectures. Vol. 1608. Lecture Notes in Computer Science. Braga, Portugal: Springer, pp. 28–115 (cited on page 80).

Ballou, Donald P. and Harold L. Pazer (1985). 'Modeling data and process quality in multi-input, multi-output information systems'. In: *Management Science* 31.2, pp. 150–162 (cited on page 94).

Balzer, Robert (1985). 'A 15 year Perspective on Automatic Programming'. In: *IEEE Transactions on Software Engineering* SE-11.11, pp. 1257–1268 (cited on page 20).

Barais, Olivier, Anne Françoise Le Meur, Laurence Duchien and Julia Lawall (2008). 'Software Architecture Evolution'. In: *Software Evolution*. Springer, pp. 233–262 (cited on page 60).

Bargas-Avila, Javier A. et al. (2010). 'Simple but Crucial User Interfaces in the World Wide Web: Introducing 20 guidelines for usable web form design'. In: *User Interfaces*. Ed. by Rita Mátrai. Rijeka, Croatia: InTech, pp. 1–10 (cited on pages 94, 97).

Beck, Kent and Cynthia Andres (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional (cited on page 8).

Beck, Kent, Mike Beedle et al. (2001). *The agile manifesto*. `http://www.agilemanifesto.org` (cited on page 8).

Beck, Kent, Ron Crocker et al. (1996). 'Industrial Experience with Design Patterns'. In: *Proceedings of the 18th International Conference on Software Engineering (ICSE)*. Berlin, Germany: IEEE Computer Society, pp. 103–114 (cited on page 50).

Beeson, Michael (1998). 'Design Principles of Mathpert: Software to Support Education in Algebra and Calculus'. In: *Computer-human interaction in symbolic computation*. Ed. by N. Kajler. Texts and Monographs in Symbolic Computation. Heidelberg, Germany: Springer, pp. 163–177 (cited on pages 77, 87).

Bennett, Judith, Fred Lubben and Sylvia Hogarth (2007). 'Bringing Science to Life: A synthesis of the research evidence on the effects of context-based and STS approaches to science teaching'. In: *Science Education* 91.3, pp. 347–370 (cited on page 43).

Benveniste, Albert, Eric Fabre, Stefan Haar and Claude Jard (2003). 'Diagnosis of Asynchronous Discrete-event Systems: a Net Unfolding Approach'. In: *IEEE Transactions on Automatic Control* 48.5, pp. 714–727 (cited on page 40).

Biegstraaten, Ton, Klaas Brink, Jan van Katwijk and Hans J. Toetenel (1994). *A Simple Railroad Controller: A Case Study in Real-time Specification*. Tech. rep. 94-86. Delft University of Technology, Department of Mathematics and Informatics (cited on pages 120–121).

Boasson, Maarten (1995). 'The Artistry of Software Architecture'. In: *IEEE Software* 12.6, pp. 13–17 (cited on page 120).

Boasson, Maarten (1996). 'Subscription as a Model for the Architecture of Embedded Systems'. In: *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems*. Montreal, Quebec, Canada: IEEE Computer Society, pp. 130–133 (cited on pages 63, 136–137, 142, 148).

Boasson, Maarten (1998). 'Software Architecture for Distributed Reactive Systems'. In: *Proceedings of the 25th Conference on Current Trends in Theory and Practice of Informatics: Theory and Practice of Informatics*. SOFSEM '98. London, UK: Springer-Verlag, pp. 1–18 (cited on pages 64, 183).

Booch, Grady (1982). 'Object-oriented Design'. In: *ACM SIGAda Ada Letters* 1.3, pp. 64–76 (cited on page 10).

Bourque, Pierre and Richard E. Fairly, eds. (2014). *Guide to the Software Engineering Body of Knowledge version 3*. IEEE Computer Society (cited on pages 11, 96).

Bouwers, Eric (2007). 'Improving Automated Feedback'. MA thesis. Utrecht University (cited on page 78).

Bower, Matt, Karen Woo, Matt Roberts and Paul Watters (2006). 'Wiki Pedagogy – A Tale of Two Wikis'. In: *Proceedings of the 7th International Conference on Information Technology Based Higher Education and Training (ITHET)*. Ultimo, NSW, Australia: IEEE Computer Society, pp. 191–202 (cited on page 167).

Brabrand, Claus, Anders Møller, Mikkel Ricky and Michael I. Schwartzbach (2000). 'Powerforms: Declarative client-side form field validation'. In: *World Wide Web* 3.4, pp. 205–214 (cited on page 93).

Brooks, Fred P. (1987). 'No Silver Bullet - Essence and Accident in Software Engineering'. In: *Computer* 20.4, pp. 10–19 (cited on page 14).

Bruning, Lucia and Berenice Michels (2013). *Concept-contextvenster*. Tech. rep. Stichting Leerplan Ontwikkeling (SLO), the Netherlands (cited on page 45).

Buck, Duane and David J. Stucki (2000). 'Design Early Considered Harmful: Graduated exposure to complexity and structure based on levels of cognitive development'. In: *ACM SIGCSE Bulletin* 32.1, pp. 75–79 (cited on page 52).

Buckley, Jim, Tom Mens, Matthias Zenger, Awais Rashid and Günter Kniesel (2005). 'Towards a Taxonomy of Software Change'. In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.5, pp. 309–332 (cited on page 14).

Burridge, Rich (1999). *Java Shared Data Toolkit User Guide*. Tech. rep. Sun Microsystems, October (cited on page 139).

Capilla, Rafael and Jan Bosch (2013). 'Binding Time and Evolution'. In: *Systems and Software Variability Management*. Springer, pp. 57–73 (cited on page 18).

Caspersen, Michael E. and Michael Kolling (2009). 'STREAM: A first programming process'. In: *ACM Transactions on Computing Education (TOCE)* 9.1, 4:1–4:29 (cited on page 93).

Caswell, Tom, Shelley Henson, Marion Jensen and David Wiley (2008). 'Open Educational Resources: Enabling universal education'. In: *International Review of Research in Open and Distance Learning* 9.1, pp. 1–11 (cited on page 159).

Chaachoua, Hamid, Jean-François Nicaud, Alain Bronner and Denis et al. Bouhineau (2004). 'Aplusix, a Learning Environment for Algebra, actual use and benefits'. In: *Proceedings of 10th International Conference on Mathematics Education (ICME-10)*. Roskilde, Denmark: IMFUFA, Department of Science, Systems and Models, Roskilde University, pp. 1–8 (cited on pages 77, 87).

Chin, Erika, Adrienne Porter Felt, Kate Greenwood and David Wagner (2011). 'Analyzing Inter-application Communication in Android'. In: *Proceedings of the 9th International Conference on Mobile systems, Applications, and Services (MobiSys)*. New York, NY, USA: ACM Digital Library, pp. 239–252 (cited on page 15).

Cîmpan, Sorana, Fabien Leymonerie and Flavio Oquendo (2005). 'Handling Dynamic Behaviour in Software Architectures'. In: *Software Architecture*. Ed. by Ron Morrison and Flavio Oquendo. Vol. 3527. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 77–93 (cited on page 60).

Claessen, Koen and John Hughes (2011). 'QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs'. In: *ACM Sigplan Notices* 46.4, pp. 53–64 (cited on page 80).

Clarke, Edmund M., E. Allen Emerson and A. Prasad Sistla (1986). 'Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2, pp. 244–263 (cited on page 149).

Clements, Paul et al. (2003). *Documenting Software Architectures, Views and beyond*. The SEI Series in Software Engineering. Addison-Wesley (cited on page 10).

Cohen, Arjeh M., Hans Cuypers, Ernesto Reinaldo Barreiro and Hans Sterk (2003). 'Interactive Mathematical Documents on the Web'. In: *Algebra, Geometry and Software Systems*. Ed. by Joswig Michael and Takayama Nobuki. Springer, pp. 289–307 (cited on pages 77, 82, 87).

Colburn, Timothy and Gary Shute (2001). 'Decoupling as a Fundamental Value of Computer Science'. In: *Minds and Machines* 21.2, pp. 241–259 (cited on page 12).

Cole, Melissa (2009). 'Using Wiki Technology to Support Student Engagement: Lessons from the trenches'. In: *Computers and Education* 52.1, pp. 141–146 (cited on page 167).

Cotroneo, Domenico, Roberto Natella, Roberto Pietrantuono and Stefano Russo (2014). 'A Survey of Software Aging and Rejuvenation Studies'. In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 10.1, 8:1–8:34 (cited on page 5).

Coulouris, George, Jean Dollimore and Tim Kindberg (1994). *Distributed Systems: Concepts and Design*. Second Edition. Addison-Wesley (cited on page 137).

Crockford, Douglas (2008). *JavaScript: The Good Parts*. O'Reilly Media, Inc. (cited on pages 22, 91).

Dahl, Ole-Johan and Kristen Nygaard (1966). 'SIMULA: An ALGOL-based Simulation Language'. In: *Communications of the ACM* 9.9, pp. 671–678 (cited on page 10).

Darwin, Ian (1999). 'GUI Development with Java'. In: *Linux Journal* 1999.61 (cited on page 52).

Davis-II, John et al. (1999). *Overview of the Ptolemy Project*. Tech. rep. ERL (cited on page 32).

Day, Christopher (1999). *Developing Teachers: The Challenges of Lifelong Learning*. London: Falmer Press, p. 249 (cited on page 158).

De Raadt, Michael, Richard Watson and Mark Toleman (2009). 'Teaching and Assessing Programming Strategies Explicitly'. In: *Proceedings of the Eleventh Australasian Conference on Computing Education*. Australian Computer Society, Inc., pp. 45–54 (cited on page 94).

Dehlinger, Josh and Jeremy Dixon (2011). 'Mobile Application Software Engineering: Challenges and research directions'. In: *Workshop on Mobile Software Engineering*. Santa Clara, California, USA. published online: `http://mobileseworkshop.org` (cited on page 15).

DeRemer, Franklin L. and Hans H. Kron (1976). 'Programming-in-the-large versus Programming-in-the-small'. In: *Programmiersprachen*. Springer, pp. 80–89 (cited on page 10).

Deursen, Arie van, Paul Klint and Joost Visser (2000). 'Domain-specific Languages: An Annotated Bibliography'. In: *ACM SIGPLAN Notices* 35.6, pp. 26–36 (cited on page 19).

Dijkstra, Edsger W. (1968). 'A Constructive Approach to the Problem of Program Correctness'. In: *BIT Numerical Mathematics* 8.3, pp. 174–186 (cited on page 9).

Dijkstra, Edsger W. (1969). 'Structured Programming'. In: *Software Engineering Techniques*. Rome, Italy: NATO Science Committee, pp. 84–88 (cited on page 9).

Dijkstra, Edsger W. (1982). 'On the Role of Scientific Thought'. In: *Selected Writings on Computing: A Personal Perspective*. Springer, pp. 60–66 (cited on pages 12, 96).

Downes, Stephen (2007). 'Models for Sustainable Open Educational Resources'. In: *The Interdisciplinary Journal of Knowledge and Learning Objects* 3, pp. 29–44 (cited on pages 160–161).

Drusinsky, Doron (2011). *Modeling and Verification Using UML Statecharts: A working guide to reactive system design, runtime monitoring and execution-based model checking*. Oxford, UK: Newnes, Elsevier (cited on page 40).

Engels, Gregor, Reiko Heckel and Stefan Sauer (2000). 'UML - A Universal Modeling Language?' In: *Application and Theory of Petri Nets 2000*. Springer, pp. 24–38 (cited on page 15).

Erev, Ido, Adi Luria and Anan Erev (2006). 'On the Effect of Immediate Feedback'. In: *Proceedings of the Chais Conference on Learning in the Technological Era*. Vol. 1. Raanana, Israel: The Open University of Israel, pp. 26–30 (cited on page 77).

Evans, Eric (2004). *Domain-driven Design: Tackling complexity in the heart of software*. Boston, United States: Addison-Wesley Professional (cited on page 30).

Evers, Joseph J.M. (1999). *DiTrans, toekomst gericht gautomatiseerd containertransport*. Tech. rep. 94/4. Delft University of Technology (cited on page 144).

Felleisen, Matthias (2001). *How to Design Programs: An introduction to programming and computing*. MIT Press (cited on pages 11, 93).

Friesen, Norman (2009). 'Open Educational Resources: New Possibilities for Change and Sustainability'. In: *International Review of Research in Open and Distance Learning* 10.5, pp. 1492–3831 (cited on page 163).

Gamma, Erich, Richard Helm, Ralph Johnsons and John Vlissides (1994). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley (cited on pages 16, 50, 56).

Garlan, David, Robert Allen and John Ockerbloom (1994). 'Exploiting Style in Architectural Design Environments'. In: *ACM SIGSOFT Software Engineering Notes* 19.5, pp. 175–188 (cited on page 61).

Garlan, David and David Notkin (1991). 'Formalizing Design Spaces: Implicit invocation mechanisms'. In: *VDM'91 Formal Software Development Methods*. Ed. by S. Prehn and Hans J. Toetenel. Vol. 551. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, pp. 31–44 (cited on page 70).

Gauthier, Richard L. and Stephen D. Ponto (1970). *Designing Systems Programs*. Prentice-Hall (cited on page 9).

Gerdes, Alex, Bastiaan J. Heeren, Johan Jeuring and Sylvia Stuurman (2008). 'Feedback Services for Exercise Assistants'. In: *Proceedings of the 7th European Conference on e-Learning (ECEL)*. Agia Napa, Cyprus: Academic Conferences Limited, pp. 402–410 (cited on page 192).

Gilbert, John K. (2006). 'On the Nature of "Contex" in Chemical Education'. In: *International Journal of Science Education* 28.9, pp. 957–976 (cited on pages 30, 43).

Gilbert, John K., Astrid M.W. Bulte and Albert Pilot (2011). 'Concept Development and Transfer in Context-Based Science Education'. In: *International Journal of Science Education* 33.6, pp. 817–837 (cited on pages 44–46).

Goguadze, Giorgi, Alberto González Palomo and Erice Melis (2005). 'Interactivity of Exercises in ActiveMath'. In: *Towards Sustainable and Scalable Educational Innovations Informed by the Learning Sciences. Sharing Good Practices of Research, Experimentation and Innovation*. Ed. by Chee-Kit Looi, David Jonassen and Mitsuru Ikeda. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 109–116 (cited on pages 77, 87).

Goldfedder, Brandon and Linda Rising (1996). 'A Training Experience with Patterns'. In: *Communications of the ACM* 39.10, pp. 60–64 (cited on page 51).

Gordon, Aaron J. (2013). 'Concepts for Mobile Programming'. In: *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. Canterbury, United Kingdom: ACM Digital Library, pp. 58–63 (cited on pages 31–32, 43).

Goudarzi, Kaveh Moazami and Jeff Kramer (1996). 'Maintaining Node Consistency in the Face of Dynamic Change'. In: *Proceedings of the Third International Conference on Configurable Distributed Systems*. Washington DC, USA: IEEE Computer Society Press, pp. 62–69 (cited on page 71).

Gregersen, Allan Raundahl and Bo Nørregaard Jørgensen (2009). 'Dynamic Update of Java Applications, Balancing change flexibility vs programming transparency'. In: *Journal of Software Maintenance and Evolution: Research and Practice* 21.2, pp. 81–112 (cited on page 60).

Gurp, Jilles van, Jan Bosch and Mikael Svahnberg (2001). 'On the Notion of Variability in Software Product Lines'. In: *Proceedings of the Working IEEE/IFIP Conference on Software*

Architecture (WICSA). Washington DC, USA: IEEE Computer Society, pp. 45–55 (cited on page 14).

Guthrie, Kevin, Rebecca Griffiths and Nancy Maron (2008). *Sustainability and Revenue Models for Online Academic Resources: An Ithaka report*. Ithaka (cited on pages 160, 162, 169).

Haas, Hugo and Allen Brown (2004). *Web Services Glossary*. W3C Working Group Note (11 February 2004) (cited on page 82).

Hattie, John and Helen Timperley (2007). 'The Power of Feedback'. In: *Review of Educational Research* 77.1, pp. 81–112 (cited on page 77).

Hazzan, Orit (2002). 'The Reflective Practitioner Perspective in Software Engineering Education'. In: *Journal of Systems and Software* 63.3, pp. 161–171 (cited on pages 13, 17).

Heeren, Bastiaan J. and Johan Jeuring (2009). 'Recognizing Strategies'. In: *Electronic Notes in Theoretical Computer Science* 237, pp. 91–106 (cited on pages 79–80).

Heeren, Bastiaan J. and Johan Jeuring (2010). 'Adapting Mathematical Domain Reasoners'. In: *Intelligent Computer Mathematics*. Springer, pp. 315–330 (cited on page 181).

Heeren, Bastiaan J. and Johan Jeuring (2014). 'Feedback Services for Stepwise Exercises'. In: *Science of Computer Programming* 88, pp. 110–129 (cited on page 76).

Heeren, Bastiaan J., Johan Jeuring, Arthur van Leeuwen and Alex Gerdes (2008). 'Specifying Strategies for Exercises'. In: *Intelligent Computer Mathematics*. Springer, pp. 430–445 (cited on page 79).

Heitkötter, Henning and Tim A. Majchrzak (2013). 'Cross-Platform Development of Business Apps with MD2'. In: *Design Science at the Intersection of Physical and Virtual Design, Proceedings of the 8th International Conference on Design Science Research in Information Systems and Technologies (DESRIST)*. Ed. by Jan vom Brocke, Riitta Hekkala, Sudha Ram and Matti Rossi. Vol. 7939. Lecture Notes in Computer Science. Helsinki, Finland: Springer, pp. 405–411 (cited on page 32).

Henderson-Sellers, Brian (2005). 'UML–the Good, the Bad or the Ugly? Perspectives from a panel of experts'. In: *Software and Systems Modeling* 4.1, pp. 4–13 (cited on page 180).

Hennecke, Martin (1999). 'Online Diagnose in Intelligenten Mathematischen Lehr-Lern-Systemen'. PhD thesis. Hildesheim University (cited on page 78).

Henzinger, Thomas A. (1996). 'The Theory of Hybrid Automata'. In: *Proceedings of the Eleventh IEEE Symposium on Logic in Computer Science (LICS)*. Washington DC, USA: IEEE Computer Society, pp. 278–292 (cited on page 149).

Henzinger, Thomas A., Xavier Nicollin, Joseph Sifakis and Sergio Yovine (1994). 'Symbolic Model Checking for Real-Time systems'. In: *Information and Computation/Information and Control* 111, pp. 193–244 (cited on page 149).

Hirschfeld, Robert, Pascal Costanza and Oscar Nierstrasz (2008). 'Context-Oriented Programming'. In: *Journal of Object Technology* 7.3, pp. 1660–1769 (cited on page 15).

Hoare, Charles Antony Richard (1965). *Record Handling*. Lecture for the NATO Summer School (cited on page 10).

Homola, Martin and Zuzanna Kubincova (2009). 'Taking Advantage of Web 2.0 in Organized Education (A Survey)'. In: *Proceedings of the International Conference on Interactive Computer Aided Learning (ICL)*. Ed. by Michael E. Auer. Villach, Austria: Kassel University Press, pp. 741–752 (cited on page 167).

Hoon, Walter A. C. A. J. de, Luc M. W. J. Rutten and Marko C. J. D. van Eekelen (1995). 'Implementing a Functional Spreadsheet in Clean'. In: *Journal of Functional Programming* 5 (03), pp. 383–414 (cited on page 9).

Hu, Minjie, Michael Winikoff and Stephen Cranefield (2012). 'Teaching Novice Programming using Goals and Plans in a Visual Notation'. In: *Proceedings of the Fourteenth Australasian*

Computing Education Conference. Vol. 123. Australian Computer Society, Inc., pp. 43–52 (cited on page 94).

Hylén, Jan (2006). 'Open Educational Resources: Opportunities and Challenges'. In: *Proceedings of Open Education 2006: Community, Culture and Content*. Paris, France: CERI, Center for Educational Research and Innovation, pp. 49–63 (cited on pages 159–160).

Jackson, Michael A. (1975). *Principles of Program Design*. Academic Press (cited on page 9).

Jeuring, Johan (2007). 'Feedback in Exercise Assistants'. In: *Book of Abstracts of Online Educa Berlin, 13th International Conference on Technology Supported Learning and Training*. Berlin, Germany: ICWE. Also published as Technical Report Utrecht University UU-CS-2007-036 (cited on page 79).

Jeuring, Johan, Harrie Passier and Sylvia Stuurman (2007). 'A Generic Framework for Developing Exercise Assistants'. In: *Proceedings of the 8th International Conference on Information Technology Based Higher Education and Training (ITHET)*. Washington DC, USA: IEEE Computer Society Press, pp. 81–91. Also available as Technical Report Utrecht University UU-CS-2007-017 (cited on pages 19, 75, 192).

Johnson, Timothy E. (1963). 'Sketchpad III: A computer program for drawing in three dimensions'. In: *Proceedings of the AFIPS Spring Joint Computer Conference*. New York, NY, USA: ACM Digital Library, pp. 347–353 (cited on page 10).

The Joint Taskforce On Computing Curricula (2013). *Computer Science Curricula 2013* (cited on page 43).

The Joint Taskforce On Computing Curricula (2009a). *Graduate Software Engineering 2009 (GSwE2009) Curriculum Guidelines for Graduate Degree Programs in Software Engineering*. Ed. by Art Pyster (cited on pages 43–44, 46).

The Joint Taskforce On Computing Curricula (2009b). *Software Engineering 2004, Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* (cited on pages 45–46).

Katwijk, Jan van, Ruud de Rooij, Sylvia Stuurman and Hans J. Toetenel (2001). 'Software Development and Verification of Dynamic Real-time Distributed Systems based on the Radio Broadcast Paradigm'. In: *Parallel and Distributed Computing Practices*. Commack, NY, USA: Nova Science Publishers, Inc., pp. 105–126 (cited on page 192).

Katwijk, Jan van and Hans J. Toetenel (1995). *Experience Using Paisley for Real-time Specification*. Tech. rep. 95-29. Delft University of Technology, Department of Technical Mathematics and Informatics (cited on pages 120, 127).

Kemerer, Chris F. (1995). 'Software Complexity and Software Maintenance: A survey of empirical research'. In: *Annals of Software Engineering* 1.1, pp. 1–22 (cited on page 21).

Kennedy, Aileen (2005). 'Models of Continuing Professional Development: A framework for analysis'. In: *Journal of In-service Education* 31.2, pp. 235–250 (cited on page 165).

Kinnunen, Päivi and Lauri Malmi (2006). 'Why Students Drop out CS1 courses'. In: *Proceedings of the Second International Workshop on Computing Education Research*. ACM, pp. 97–108 (cited on page 91).

Kirschner, Paul A., John Sweller and Richard E Clark (2006). 'Why Minimal Guidance during Instruction does Not Work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching'. In: *Educational psychologist* 41.2, pp. 75–86 (cited on pages 13, 45, 91).

Ko, Minhyuk, Yong-Jin Seo, Bup-Ki Min, Seunghak Kuk and Hyeon Soo Kim (2012). 'Extending UML Meta-model for Android Application'. In: *Proceedings of the 2012 IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS)*. Shanghai, China: IEEE Computer Society, pp. 669–674 (cited on page 32).

Kou, Siming, Muhammad Ali Babar and Amit Sangroya (2010). 'Modeling Security for Service Oriented Applications'. In: *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. ECSA '10. Copenhagen, Denmark: ACM, pp. 294–301 (cited on page 41).

Kraemer, Frank A. (2011). 'Engineering Android Applications Based on UML Activities'. In: *Proceedings of the 14th Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ed. by Jon Whittle, Tony Clark and Thomas Kühne. Vol. 6981. Lecture Notes in Computer Science. Wellington, New Zealand: Springer, pp. 183–197 (cited on pages 32, 36, 39).

Kramer, Jeff (2007). 'Is Abstraction the Key to Computing?' In: *Communications of the ACM* 50.4, pp. 36–42 (cited on pages 12–13, 96).

Kramer, Jeff and Jeff Magee (1990). 'The Evolving Philosophers Problem: Dynamic change management'. In: *IEEE Transactions on Software Engineering* 16.11, pp. 1293–1306 (cited on pages 70, 121).

Kruchten, Philippe (1995). 'Architectural Blueprints: The 4+1 View Model of Architecture'. In: *IEEE Software* 12.6, pp. 42–50 (cited on pages 10, 61).

Kruchten, Philippe (2004). *The Rational Unified Process: an introduction*. Addison-Wesley Professional (cited on page 180).

Kurshan, Robert P. (1997). 'Formal Verification in a Commercial Setting'. In: *Proceedings of the 34th Design Automation Conference*. Anaheim, California, USA: ACM Digital Library, pp. 258–262 (cited on page 148).

Larman, Craig (2012). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Third Edition. Upper Saddle River, United States: Pearson Education (cited on page 30).

Lehman, Meir M. and Les A. Belady (1985). *Program Evolution - processes of software change*. London: Academic Press (cited on page 6).

Lehman, Meir M., Dewayne E. Perry and Juan F. Ramil (1998). 'On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution'. In: *Proceedings of the Fifth International Software Metrics Symposium*. Bethesda, Maryland, USA: IEEE Computer Society Press, pp. 84–88 (cited on pages 6–7).

Li, Zhen and Eileen Kraemer (2013). 'Programming with Concurrency: Threads, Actors, and Coroutines'. In: *Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. Boston, Massachusetts, United States: IEEE Computer Society Press, pp. 1304–1311 (cited on page 41).

Libbrecht, Paul and Stefan Winterstein (2005). 'The Service Architecture in the ActiveMath Learning Environment'. In: *First International Kaleidoscope, Learning Grid SIG Workshop on Distributed e-Learning Environments*. Naples, Italy: Technology Enhanced Learning European Advanced Research Consortium (TELEARC). published online at `http://ewic.bcs.org/category/16635` (cited on page 88).

Liskov, Barbara and Stephen Zilles (1974). 'Programming with Abstract Data Types'. In: *ACM Sigplan Notices* 9.4, pp. 50–59 (cited on page 9).

Liu, Tongping, Charlie Curtsinger and Emery D. Berger (2011). 'DThreads: Efficient Deterministic Multithreading'. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. Cascais, Portugal: ACM Digital Library, pp. 327–336 (cited on page 41).

Lodder, Josje S., Johan Jeuring and Harrie J.M. Passier (2006). 'An Interactive Tool for Manipulating Logical Formulae'. In: *Proceedings of the Second International Conference on Tools for Teaching Logic (TICTTL)*. Ed. by M. Manzano, B. Pérez Lancho and A. Gil. Salamanca, Spain: Springer (cited on page 87).

Lodder, Josje S., Harrie J.M. Passier and Sylvia Stuurman (2008). 'Using IDEAS in Teaching Logic, Lessons Learned'. In: *Proceedings of the International Conference on Computer Science and Software Engineering (CSSE)*. Vol. 5. Wuhan, China: IEEE Computer Society Press, pp. 553–557. Also available as Technical Report Utrecht University UU-CS-2006-040 (cited on pages 19, 76, 192).

Lodderstedt, Torsten, David Basin and Jürgen Doser (2002). 'SecureUML: A UML-based modeling language for model-driven security'. In: *UML 2002, The Unified Modeling Language*. Ed. by Jean-Marc Jézéquel, Heinrich Hussmann and Stephen Cook. Vol. 2460. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 426–441 (cited on page 41).

Lüer, Chris, David S. Rosenblum and André van der Hoek (2001). 'The Evolution of Software Evolvability'. In: *Proceedings of the 4th International Workshop on Principles of Software Evolution*. IWPSE '01. Vienna, Austria: ACM Digital Library, pp. 134–137 (cited on pages 6, 17).

Lutje Spelberg, Ronald F., Sylvia Stuurman and Hans J. Toetenel (1996). *MTCCS*. Tech. rep. Delft University of Technology, Faculty of Technical Mathematics and Informatics (cited on page 193).

Ma, Linxiao, John Ferguson, Marc Roper and Murray Wood (2007). 'Investigating the Viability of Mental Models held by Novice Programmers'. In: *ACM SIGCSE Bulletin* 39.1, pp. 499–503 (cited on page 91).

Magee, Jeff and Jeff Kramer (1996). 'Dynamic Structure in Software Architectures'. In: *ACM SIGSOFT Software Engineering Notes* 21.6, pp. 3–14 (cited on page 71).

Magee, Jeff and Jeff Kramer (2006). *State Models and Java Programs*. Second Edition. Chichester, England: Wiley (cited on page 40).

Marchetto, Alessandro, Paolo Tonella and Filippo Ricca (2008). 'State-based Testing of Ajax Web Applications'. In: *Proceedings of the 2008 1st International Conference on Software Testing, Verification, and Validation*. Lillehammer, Norway: IEEE Computer Society, pp. 121–130 (cited on page 40).

Mayr, Ernst (1997). 'The Objects of Selection'. In: *Proceedings of the National Academy of Sciences* 94.6, pp. 2091–2094 (cited on page 6).

McConnell, Steve (2004). *Code Complete*. Microsoft Press (cited on page 11).

McCracken, Michael et al. (2001). 'A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students'. In: *ACM SIGCSE Bulletin* 33.4, pp. 125–180 (cited on page 90).

McGreal, Rory (2004). 'Learning Objects: A practical definition'. In: *International Journal of Instructional Technology and Distance Learning (IJITDL)* 9.1. available online at `http://www.itdl.org/Journal/Sep_04/article02.htm` (cited on page 159).

Mehta, Nikunj R., Nenad Medvidovic and Sandeep Phadke (2000). 'Towards a Taxonomy of Software Connectors'. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. Limerick, Ireland: ACM Digital Library, pp. 178–187 (cited on page 10).

Merriam-Webster (2004). *Merriam-Webster's Collegiate Dictionary*. Merriam-Webster (cited on page 8).

Merriënboer, Jeroen J.G. van, Richard E. Clark and Marcel B.M. de Croock (2002). 'Blueprints for Complex Learning: The 4C/ID-model'. In: *Educational Technology Research and Development* 50.2, pp. 39–61 (cited on page 92).

Merriënboer, Jeroen J.G. van and Paul A. Kirschner (2001). 'Three Worlds of Instructional Design: State of the art and future directions'. In: *Instructional Science* 29.4-5, pp. 429–441 (cited on page 45).

Merriënboer, Jeroen J.G. van and Paul A. Kirschner (2013). *Ten Steps to Complex Learning, a systematic appraoch to four-component instructional design*. Second Edition. New York, NY, USA: Taylor & Francis (cited on page 91).

Merrill, M. David (2002). 'First Principles of Instruction'. In: *Educational technology research and development* 50.3, pp. 43–59 (cited on pages 13, 45, 91).

Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Second Edition. Upper Saddle River, New Jersey, USA: Prentice Hall (cited on page 97).

Michels, Gerard, Sebastiaan Joosten, Jaap van der Woude and Stef Joosten (2011). 'Ampersand'. In: *Proceedings of the 12th International Conference on Relational and Algebraic Methods in Computer Science (RAMICS)*. Ed. by Harrie Swart. Vol. 6663. Lecture Notes in Computer Science. Rotterdam, the Netherlands: Springer, pp. 280–293 (cited on page 20).

Microsystems, Sun (1998). *The JavaSpaces Specification*. Tech. rep. published online at `http://www.oracle.com/technetwork/articles/javase/javaspaces-140665.html`. Sun Microsystems (cited on page 138).

Minto, Barbara (2009). *The pyramid principle: logic in writing and thinking*. Pearson Education (cited on page 190).

Morrison, Gary R., Steven M. Ross, Mala Gopalakrishnan and Jason Casey (1995). 'The Effects of Feedback and Incentives on Achievement in Computer-based Instruction'. In: *Contemporary Educational Psychology* 20.1, pp. 32–50 (cited on page 77).

Mory, Edna H. (2003). 'Feedback Research Revisited'. In: *Handbook of Research for Educational Communications and Technology*. Ed. by D.H. Jonassen. Heidelberg, Germany: Springer, pp. 745–785 (cited on page 77).

Mouheb, Djedjiga et al. (2009). 'Weaving Security Aspects into UML 2.0 Design Models'. In: *Proceedings of the 13th Workshop on Aspect-oriented Modeling*. AOM '09. Charlottesville, Virginia, USA: ACM, pp. 7–12 (cited on page 41).

Myagmar, Suvda, Adam J. Lee and William Yurcik (2005). 'Threat Modeling as a Basis for Security Requirements'. In: *Symposium on Requirements Engineering for Information Security (SREIS)*, pp. 1–8 (cited on page 41).

Nielsen, Jacob (2006). *Participation Inequality: Encouraging more users to contribute*. Alertbox Columns, `http://www.nngroup.com/articles/participation-inequality/` (cited on page 172).

Nudelman, Greg (2013). *Android Design Patterns: Interaction Design Solutions for Developers*. Indianapolis, United States: John Wiley & Sons (cited on page 42).

Odersky, Martin, Lex Spoon and Bill Venners (2010). *Programming in Scala*. Second Edition. Artima (cited on pages 22, 163).

Oliveira, Paulo, Fátima Rodrigues and Pedro Rangel Henriques (2005). 'A Formal Definition of Data Quality Problems'. In: *International Conference on Information Quality ICIQ*. Cambridge, Massachusstes, USA: MIT Information Quality (MITIQ), pp. 14–24 (cited on page 94).

Oram, Andy and Greg Wilson (2007). *Beautiful Code: Leading programmers explain how they think*. O'Reilly Media, Inc (cited on page 89).

Oreizy, Peyman, Nenad Medvidovic and Richard N. Taylor (1998). 'Architecture-based Runtime Software Evolution'. In: *Proceedings of the 20th International Conference on Software Engineering (ICSE)*. Long Beach, California, USA: IEEE Computer Society Press, pp. 177–186 (cited on page 71).

Parada, Abilio G and Lisane B. de Brisolara (2012). 'A Model Driven Approach for Android Applications Development'. In: *Proceedings of the Conference on Computing System Engineering, Brazilian Symposium (SBESC)*. IEEE Computer Society. Natal, Brazil: IEEE Computer Society, pp. 192–197 (cited on pages 32, 36).

Parnas, David Lorge (1985). 'Software Aspects of Strategic Defense Systems'. In: *Communications of the ACM* 28.12, pp. 1326–1335 (cited on page 20).

Parnas, David Lorge (1994). 'Software Aging'. In: *Proceedings of the 16th International Conference on Software Engineering (ICSE)*. Sorrento, Italy: IEEE Computer Society Press, pp. 279–287 (cited on page 5).

Parnas, David Lorge, Paul C. Clements and David M. Weiss (1985). 'The Modular Structure of Complex Systems'. In: *IEEE Transactions on Software Engineering* SE-11.3, pp. 259–266 (cited on page 10).

Parnas, David Lorge, A. John van Schouwen and Shu Po Kwan (1990). 'Evaluation of Safety-critical Software'. In: *Communications of the ACM* 33.6, pp. 636–648 (cited on page 121).

Passier, Harrie J.M. and Johan Jeuring (2006). 'Feedback in an Interactive Equation Solver'. In: *Proceedings of the Web Advanced Learning Conference and Exhibition (WebALT)*. Ed. by M. Seppälä, S. Xambo and O. Caprotti. Oy WebALT, pp. 53–68 (cited on page 87).

Passier, Harrie J.M., Sylvia Stuurman and Harold Pootjes (2014). 'Beautiful Javascript: How to guide students to create good and elegant code'. In: *Proceedings of the Fourth Computer Science Education Research Conference (CSERC)*. Berlin, Germany: ACM Digital Library, pp. 65–76 (cited on page 192).

Perry, Dewayne E. and Alexander L. Wolf (1992). 'Foundations for the Study of Software Architecture'. In: *ACM SIGSOFT Software Engineering Notes* 17.4, pp. 40–52 (cited on page 10).

Peyton Jones, Simon L. (2003). *Haskell 98 Language and Libraries: the revised report*. Cambridge University Press (cited on page 80).

Pietraszek, Tadeusz and Chris Vanden Berghe (2006). 'Defending against Injection Attacks through Context-Sensitive String Evaluation'. In: *Recent Advances in Intrusion Detection*. Vol. 3858. Lecture Notes in Computer Science. Springer, pp. 124–145 (cited on page 94).

Pina, Luıs and Michael Hicks (2013). 'Rubah: Efficient, general-purpose dynamic software updating for Java'. In: *Proceedings of the Workshop on Hot Topics in Software Upgrades (HotSWUp)*. available online at `https://www.usenix.org/conference/hotswup13/workshop-program` (cited on page 60).

Postel, Jon (1981). *Request For Comments 793-Transmission Control Protocol* (cited on pages 94, 97).

Proulx, Viera K. (2000). 'Programming Patterns and Design Patterns in the Introductory Computer Science Course'. In: *ACM SIGCSE Bulletin* 32.1, pp. 80–84 (cited on page 51).

Rahn, Mirko and Johannes Waldmann (2002). 'The Leipzig Autotool System for Grading Student Homework'. In: *Workshop on Functional and Declarative Programming in Education (FDPE)*. Kiel, Germany: University of Kiel (cited on page 87).

Rajkumar, Raganuthan and Mike Gagliardi (1996). 'High Availability in The Real-Time Publisher/Subscriber Inter-Process Communication Model'. In: *Proceedings 17th IEEE Real-Time Systems Symposium*. IEEE Computer Society Technical Committee on Real-Time Systems. Washington, DC: IEEE Computer Society Press, pp. 136–141 (cited on page 138).

Richards, Gregor, Sylvain Lebresne, Brian Burg and Jan Vitek (2010). 'An Analysis of the Dynamic Behavior of JavaScript Programs'. In: *ACM Sigplan Notices* 45.6, pp. 1–12 (cited on page 91).

Riley, Derek (2012). 'Using Mobile Phone Programming to Teach Java and Advanced Programming to Computer Scientists'. In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE)*. Raleigh, North Carolina, USA: ACM Digital Library, pp. 541–546 (cited on pages 32, 36).

Rooij, Ruud C.M. de (1998). 'Subscription-based Communication for Distributed Embedded Java Applications'. In: *Proceedings of the Fourth Annual Conference of the Advanced School*

for Computing and Imaging (ASCI). Lommel, Belgium: Advanced School for Computing and Imaging (ASCI) (cited on pages 66, 139, 148).

Royce, Winston W. (1987). 'Managing the Development of Large Software Systems: Concepts and techniques'. In: *Proceedings of the 9th International Conference on Software Engineering (ICSE*. Los Alamitos CA, USA: IEEE Computer Society Press, pp. 328–338 (cited on page 8).

Rozanski, Nick and Eoin Woods (2005). *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional (cited on page 10).

Rumbaugh, James R., Michael R. Blaha, William Lorensen, Frederick Eddy and William Premerlani (1990). *Object-oriented Modeling and Design*. Prentice-hall (cited on page 10).

Rumbaugh, James, Ivar Jacobson and Grady Booch (2004). *Unified Modeling Language Reference Manual, The*. Pearson Higher Education (cited on page 10).

Schön, Donald A. (1987). *Educating the Reflective Practitioner*. San Francisco: Jossey-Bass (cited on page 13).

Schuwer, Robert, Andrew Lane, Anda Counotte-Potman and Martina Wilson (2011). 'A Comparison of Production Processes for OER'. In: *Open Courseware Consortium Global Meeting*. Cambridge, Massachusetts, USA: Open Courseware Consortium. published online at `http://oro.open.ac.uk/29817/` (cited on page 168).

Schuwer, Robert and Fred Mulder (2009). 'OpenER, a Dutch initiative in Open Educational Resources'. In: *Open Learning: The Journal of Open and Distance Learning* 24.1, pp. 67–76 (cited on pages 158, 162).

Segal, Mark E. and Ophir Frieder (1988). 'Dynamic Program Updating in a Distributed Computer System'. In: *Proceedings of the Conference on Software Maintenance*. Phoenix, Arizona: IEEE Computer Society Press, pp. 198–203 (cited on pages 71, 121).

Seifzadeh, Habib, Hassan Abolhassani and Mohsen Sadighi Moshkenani (2013). 'A Survey of Dynamic Software Updating'. In: *Journal of Software: Evolution and Process* 25.5, pp. 535–568 (cited on page 17).

Shalloway, Alan and James Trott (2005). *Design Patterns Explained: A new perspective on object-oriented design*. Addison-Wesley Professional (cited on page 50).

Shaw, Mary (1995). 'Beyond Objects: A software design paradigm based on process control'. In: *ACM SIGSOFT Software Engineering Notes* 20.1, pp. 27–38 (cited on page 127).

Shaw, Mary (1996). 'Some Patterns for Software Architectures'. In: *Pattern Languages of Program Design 2*. Addison-Wesley, pp. 255–269 (cited on page 13).

Shaw, Mary and Paul Clements (1997). 'A Field Guide to Boxology: Preliminary classification of architectural styles for software systems'. In: *Proceedings of The 21st Annual International Computer Software and Applications Conference (COMPSAC)*. Washington, DC, USA: IEEE Computer Society Press, pp. 6–13 (cited on page 120).

Shaw, Mary and David Garlan (1996). *Software Architecture: perspectives on an emerging discipline*. Prentice Hall (cited on pages 10, 120, 122).

Singhal, Amit (2004). 'Challenges in Running a Commercial Search Engine'. In: *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Salvador, Brazil: ACM, pp. 432–432 (cited on page 7).

Song, Hui et al. (2011). 'Supporting Runtime Software Architecture: A bidirectional transformation based approach'. In: *Journal of Systems and Software* 84.5, pp. 711–723 (cited on page 60).

Stankovic, John A. (1996). 'Real-time and Embedded Systems'. In: *ACM Computing Surveys (CSUR)* 28.1, pp. 205–208 (cited on pages 61, 131).

Steels, Luc (2003). 'Intelligence with Representation'. In: *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 361.1811, pp. 2381–2395 (cited on page 183).

Stefanov, Stoyan (2010). *JavaScript patterns*. Sebastopol, California, USA: O'Reilly Media (cited on page 98).

Störrle, Harald and J.H. Hausmann (2004). 'Semantics of UML 2.0 Activities'. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Rome, Italy: IEEE Computer Society, pp. 235–242 (cited on page 37).

Stringfellow, Catherine and Divya Mule (2013). 'Smartphone Applications As Software Engineering Projects'. In: *Journal of Computing Sciences in Colleges* 28.4, pp. 27–34 (cited on pages 32, 36).

Stuurman, Sylvia and Bastiaan J. Heeren (2012). *Scala Opener Cursus (in Dutch)*. `http://portal.ou.nl/web/topic-scala` (cited on page 164).

Stuurman, Sylvia (1997). *Evolving Software Architectures: A Position Paper*. Presented at the Living with Inconsistency Workshop in conjunction with ICSE (cited on pages 60–61, 193).

Stuurman, Sylvia (1999). 'Software Architecture and Java Beans'. In: *Proceedings of the First working IFIP Conference on Software Architecture (WICSA)*. IFIP Conference Proceedings. Washington DC, USA: IEEE Computer Society Press, pp. 183–199 (cited on pages 15, 180, 193).

Stuurman, Sylvia, Marko C.J.D. van Eekelen and Bastiaan J. Heeren (2012). 'A New Method for Sustainable Development of Open Educational Resources'. In: *Proceedings of Second Computer Science Education Research Conference (CSERC)*. Wroclaw, Poland: ACM Digital Library, pp. 57–66 (cited on page 192).

Stuurman, Sylvia and Gert Florijn (2004). 'Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education'. In: vol. 36. SIGCSE Bulletin 3. Leeds, United Kingdom: ACM, pp. 151–155 (cited on page 192).

Stuurman, Sylvia and Johan Jeuring (2007). *Turning an Interactive Tool implemented in Haskell into a Web Application – an experience report*. Technical report Universiteit Utrecht (cited on page 192).

Stuurman, Sylvia and Jan van Katwijk (1997). 'Evaluation of Software Architectures for a Control System: a case study'. In: *Proceedings of the Second International Conference on Coordination Languages and Models (COORDINATION)*. Heidelberg, Germany: Springer, pp. 157–171 (cited on page 193).

Stuurman, Sylvia and Jan van Katwijk (1998). 'On-line Change Mechanisms: the Software Architectural Level'. In: *Proceedings of the 6th International Symposium on the Foundations of Software Engineering*. Orlando, Florida, US: ACM Digital Library, pp. 80–86 (cited on pages 145, 193).

Stuurman, Sylvia, Harrie J.M. Passier and Bernard E. van Gastel (2014). 'The Design of Mobile Apps: Which Modeling Techniques to Teach?' In: *Proceedings of the Fourth Computer Science Education Research Conference (CSERC)*. Berlin, Germany: ACM Digital Library, pp. 93–100 (cited on page 192).

Stuurman, Sylvia, Frank J. Wester and Manuela Witsier-Voglet (2002). *Design Patterns*. Open Universiteit Nederland (cited on page 50).

Swierstra, S. Doaitse and Pablo R. Azero Alcocer (1999). 'Fast, Error Correcting Parser Combinators: A short tutorial'. In: *Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM)*. Vol. 1725. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, pp. 111–129 (cited on page 78).

Taylor, Richard N. et al. (1996). 'A Component-and Message-based Architectural Style for GUI Software'. In: *IEEE Transactions on Software Engineering* 22.6, pp. 390–406 (cited on page 71).

TIBCO (1997). *TIB/Rendezvous White Paper* (cited on page 138).

Toetenel, Hans J., Ronald F. Lutje Spelberg, Sylvia Stuurman and Jan van Katwijk (1996). 'Modeling and Analysis of Complex Computer Systems-the MTCCS Approach'. In: *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. Montreal, Quebec, Canada: IEEE Computer Society Press, pp. 423–430 (cited on pages 20, 117, 181, 193).

Wallingford, Eugene (1996). 'Toward a First Course Based on Object-oriented Patterns'. In: *ACM SIGCSE Bulletin* 28.1, pp. 27–31 (cited on page 51).

Whalley, Jacqueline L. et al. (2006). 'An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies'. In: *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., pp. 243–252 (cited on page 91).

Whitelegg, Elizabeth and Malcolm Parry (1999). 'Real-life Contexts for Learning Physics: meanings, issues and practice'. In: *Physics Education* 34.2, pp. 68–72 (cited on page 43).

Whittaker, Steve, Loren Terveen, Will Hill and Lynn Cherny (1998). 'The Dynamics of Mass Interaction'. In: *Proceedings of the 1998 ACM Conference on Computer Cupported Cooperative Work (CSCW)*. Seattle, Washington, United States: ACM Digital Library, pp. 257–264 (cited on page 172).

Wilkinson, Nancy M. (1998). *Using CRC Cards: An informal approach to object-oriented development*. Vol. 6. Cambridge University Press (cited on page 10).

Williams, Byron J. and Jeffrey C. Carver (2010). 'Characterizing Software Architecture Changes: A systematic review'. In: *Information and Software Technology* 52.1, pp. 31–51 (cited on page 7).

Wirth, Niklaus (1971). 'Program Development by Stepwise Refinement'. In: *Communications of the ACM* 14.4, pp. 221–227 (cited on page 9).

Wissen, Bart van, Nicholas Palmer, Roelof Kemp, Thilo Kielmann and Henri Bal (2010). 'ContextDroid: An expression-based context framework for Android'. In: *Proceedings of the International Workshop on Sensing for App Phones (PhoneSense)*. Zürich, Switzerland: ACM Digital Library, pp. 1–5. available online at `http://sensorlab.cs.dartmouth.edu/phonesense/proceeding.pdf` (cited on page 15).

Witt, Bernard I., F. Terry Baker and Everett W. Merritt (1993). *Software Architecture and Design: Principles, models, and methods*. John Wiley & Sons, Inc. (cited on page 61).

Würthinger, Thomas, Christian Wimmer and Lukas Stadler (2010). 'Dynamic Code Evolution for Java'. In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. ACM, pp. 10–19 (cited on page 60).

Yang, Shengqian, Dacong Yan and Atanas Rountev (2013). 'Testing for Poor Responsiveness in Android applications'. In: *Proceedings of the 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*. IEEE Computer Society. San Francisco, USA: IEEE Computer Society, pp. 1–6 (cited on page 41).

You, Juan, Junquan Li and Song Xia (2012). 'A Survey on Formal Methods Using in Software Development'. In: *Proceedings of the IET International Conference on Information Science and Control Engineering (ICISCE)*. Shenzhen, China: IET, pp. 1–4 (cited on page 20).

Young, Matt and Taner Edis (2006). *Why Intelligent Design fails: A scientific critique of the new creationism*. Rutgers University Press (cited on page 6).

Yourdon, Edward and Larry L. Constantine (1979). *Structured Design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc. (cited on page 12).
Zinn, Claus (2006). 'Feedback to Student Help Requests and Errors in Symbolic Differentiation'. In: *Proceedings of the 8th International Conference on Intelligent Tutoring Systems (ITS)*. Vol. 4043. Lecture Notes in Computer Science. Jhongli, Taiwan: Springer, pp. 349–359 (cited on page 79).

Sylvia Stuurman studied Biology and graduated in Computer Science. She works as assistant professor at the Open University of the Netherlands.

Design for Change addresses the need to optimise the design of software with respect to future changes. Both technical and educational aspects are discussed.

ISBN 978 94 92231 11 6

6315221